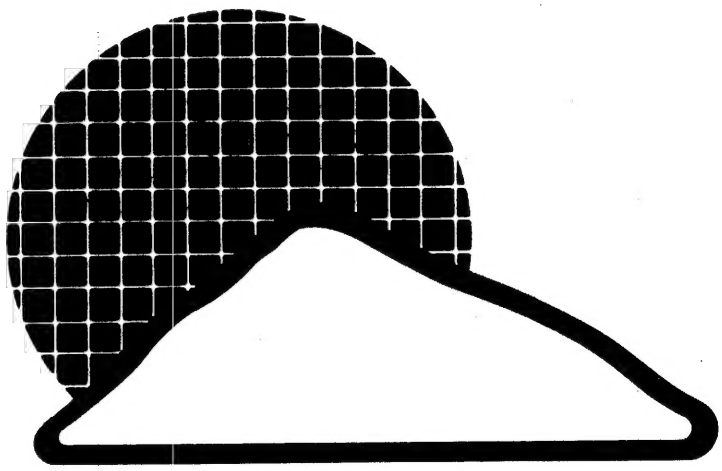
EK-DCJ11-UG-PRE

DCJ11 Microprocessor User's Guide



PRELIMINARY

digital

EK-DCJ11-UG-PRE

DCJ11 Microprocessor User's Guide

PRELIMINARY

Prepared by Educational Services
of
Digital Equipment Corporation

Preliminary, October 1983

The material in this manual is for informational purposes and is subject to change without notice.

Digital Equipment Corporation assumes no responsibility for any errors which may appear in this manual.

Printed in U.S.A.

The following are trademarks of Digital Equipment Corporation:

DEC	EduSystem	RSTS
DECnet	IAS	RSX
DECUS	MASSBUS	TOPS-10
DECsystem-10	MI' 'C-11	TOPS-20
DECSYSTEM-20	OM. 'BUS	UNIBUS
DECwriter	OS/8	VAX
DIBOL	PDP	VMS
digital	PDT	VT

CONTENTS

FIGURES

TABLES

This user's guide is intended to familiarize the reader with the hardware and software characteristics of the DCJ11 microprocessor CPU chip. It is assumed that the reader has had some experience with microprocessor design. Readers should also have some familiarity with PDP-11 architecture.

The book is organized as follows:

Chapter 1 provides an architectural overview of the DCJ11.

Chapter 2 describes the function of each DCJ11 pin.

Chapter 3 describes the various types of DCJ11 bus cycles and provides an overview of the timing relationships among DCJ11 inputs and outputs during these cycles.

Chapter 4 describes the architecture and operation of the DCJ11's integral memory management unit.

Chapter 5 provides information on three special features integral to the DCJ11: cache memory registers (this description also includes cache memory design considerations), console ODT (also called micro-ODT), and pipeline processing.

Chapter 6 describes the DCJ11 base instruction set.

Chapter 7 describes the integral floating-point unit and its instruction set.

Chapter 8 provides some introductory information on interfacing external logic to the DCJ11. Power-up and initialization circuits are provided.

Appendix A contains a summary of the DCJ11 DC characteristics.

Appendix B contains a summary of the DCJ11 AC characteristics.

Appendix C summarizes the hardware differences between: (1) the DCJ11 and the PDP-11/44 and (2) the DCJ11 and the PDP-11/70. Appendix C also contains a summary of the software differences between the DCJ11 and other processors in the PDP-11 family.

Appendix D describes how to determine the duration of a DCJ11 instruction. Timings for both the base instruction set and the floating-point instruction set are provided.

Appendix E contains a brief glossary of some DCJ11 terms.

1.1 INTRODUCTION

This chapter provides a brief introduction to the architecture of the DCJ11 microprocessor. The DCJ11 is organized as shown in Figure 1-1.

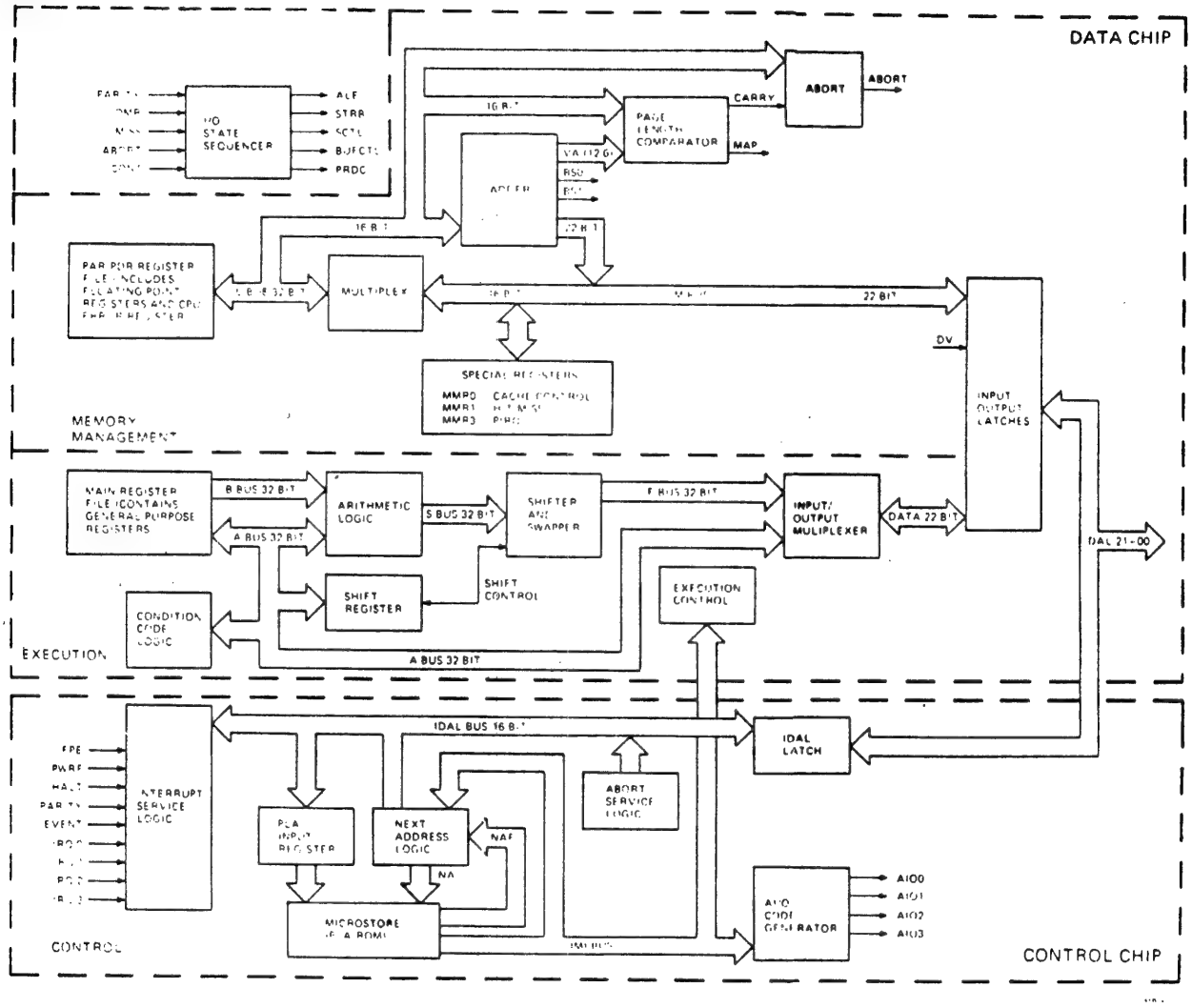


Figure 1-1 DCJ11 Block Diagram

As shown in Figure 1-1, the DCJ11 microprocessor consists of a data chip and a control chip.

The data chip performs all arithmetic and logic functions, handles all data and address transfers, and generates most of the signals used for system timing. In addition to the primary execution data path, the data chip contains memory management logic, an I/O state sequencer, and floating-point and cache control registers.

The control chip directs the operation of the data chip with microinstructions. The major components of the control chip are the microprogram control store and the microprogram sequencing logic.

A detailed description of the data chip and control chip and the interface between them is beyond the scope of this book. We will consider the data chip and control chip as one functional unit and will describe only those portions of this unit that are architecturally significant to the design engineer.

The remainder of this chapter briefly describes each of the major components of the DCJ11 architecture. The chapter covers six major topics:

- o General-purpose registers
- o Processor status word
- o Traps and interrupts
- o Floating point processing
- o Memory system registers
- o DMA mechanism

1.2 GENERAL-PURPOSE REGISTERS

As shown in Figure 1-2, the DCJ11 has a dual set of six registers R0 through R5 and R0' through R5', three stack pointers (R6) corresponding to the three processor modes (see Paragraph 1.3.1), and a program counter (R7). R0 through R5 is also referred to as register set 0 and R0' through R5' is also called register set 1.

These registers are called general-purpose because they can be used in a variety of ways. General-purpose registers serve as accumulators, index registers, autoincrement registers, autodecrement registers, or as stack pointers for temporary storage of data. Arithmetic operations can be performed between one general-purpose register and another or between a general-purpose register and memory or an I/O device register.

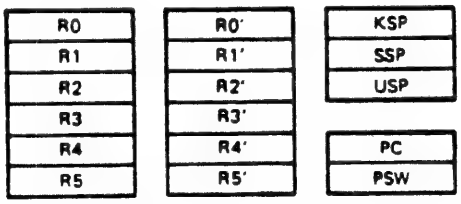


Figure 1-2 DCJ11 General-Purpose Registers

At any given time, either register set R0 through R5 is used or register set R0' through R5' is used. The two sets can not be used simultaneously. These general-purpose registers are organized as two sets to increase the speed of context switching and some types of real-time data handling.

Register R6 is used as the hardware stack pointer (SP), which indicates the last entry in the appropriate stack (the stacks are common temporary areas with LIFO - last in, first out - characteristics). There are three stack pointers: a kernel stack pointer (KSP), a supervisor stack pointer (SSP), and a user stack pointer (USP). Each stack pointer is associated with a different processor mode (see Paragraph 1.3.1). When an interrupt or trap occurs, the current CPU state (PC and PS) is automatically pushed on the stack indicated by the interrupt or trap vector (see Paragraph 1.4 for more information on interrupts and traps). The stack-based architecture also facilitates reentrant programming.

Register R7 is used as the program counter (PC). The PC contains the address of the next instruction to be executed; thereby controlling the order of execution of instructions. The PC is a general-purpose register in the sense that it is directly accessible by all single- and double-operand instructions. Much of the power of the DCJ11 instruction set is achieved by utilizing the PC in conjunction with various addressing modes. The PC is not normally used as an accumulator for arithmetic operations.

1.3 PROCESSOR STATUS WORD

As shown in Figure 1-3, the processor status word (PS) contains the condition codes describing the arithmetic or logical results of the last instruction, a trace bit that forces a trap at the end of an instruction (used for program debugging), the current processor priority, and the current and previous processor modes. The PS is located at physical address 17777776.

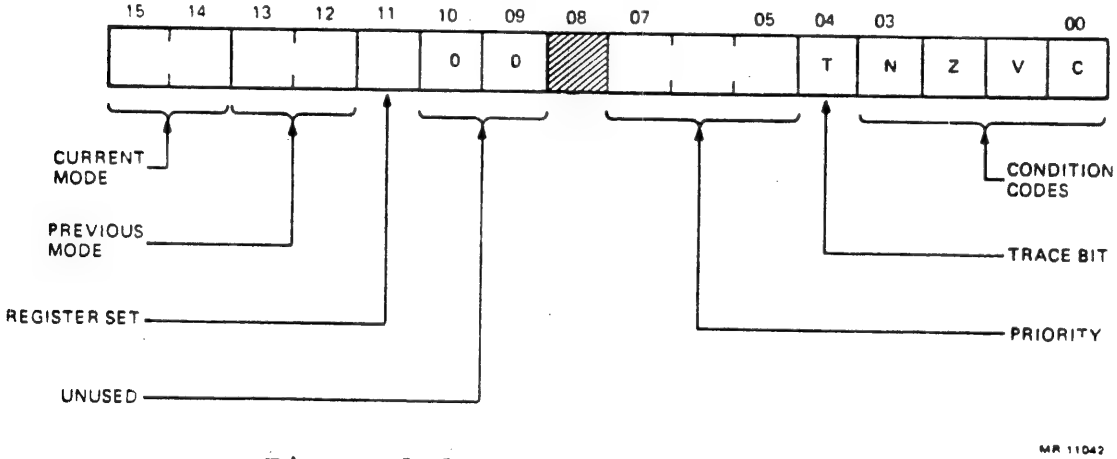


Figure 1-3 Processor Status Word

BIT	NAME	FUNCTION
15:14	Current Mode (RW, protected)	Current processor mode:

Bits 15	Bits 14	Mode
0	0	Kernel
0	1	Supervisor
1	0	Illegal
1	1	User

13:12	Previous Mode (RW, protected)	Previous processor mode; same encoding as for bits <15:14>.
11	Register Set (RW, protected)	General register set select: 0 = register set 0 (R0--R5). 1 = register set 1 (R0'--R5').
10:9	Unused (Read only)	The bits are unused and are always read as zeroes.
8	Reserved (RW)	This bit is reserved for future DIGITAL use.
7:5	Priority (RW, protected)	Processor interrupt priority level:

Bits			Priority Level
7	6	5	
1	1	1	7
1	1	0	6
1	0	1	5
1	0	0	4
0	1	1	3
0	1	0	2
0	0	1	1
0	0	0	0

4	Trace Trap (RW, protected)	Also called the T-bit. When set, the processor traps to location 14 at the end of the current instruction. This bit cannot be set directly by writing data to the PS. This bit is typically set by the RTI/RTT instruction. Trace trap is disabled when this bit is zero.
3:0	Condition Codes (RW)	Processor condition codes: N: Set if the result of the previous operation was negative. Z: Set if the result of the previous operation was zero. V: Set if the previous operation resulted in an ar_thmetic overflow. C: Set if the previous operation resulted in a carry of its most significant bit.

1.3.1 Processor Modes - Three processor modes (user, supervisor, and kernel) permit a fully protected environment for a multiprogramming system by providing the programmer with three distinct sets of processor stacks and memory management registers for memory mapping. In addition, certain PDP-11 instructions are privileged in that their operation is inhibited in supervisor and user modes. For example, in supervisor or user mode, the processor will ignore the RESET and SPL (Set Priority Level) instructions and the HALT instruction will cause a trap through the vector at virtual address 4 in kernel data space. In kernel mode, the processor will execute all instructions. A summary of the effects of processor modes on various instruction types is provided in Table 1-1.

Table 1-1 Instructions Influenced by Processor Modes

Instruction or Instruction Type	Operation in Kernel Mode	Operation in Supervisor/User Mode
HALT	Depends on halt option selected (see Paragraph 1.5)	Traps through a vector at location 4 in kernel data space.
WAIT, RESET, SPL	Executes as specified	Executes as a NOP.
RTI, RTT, MPTS	Can alter PS<7:5>	Can not alter PS<7:5>
Stack Reference	Checked for stack overflow.	Not checked for stack overflow.

1.3.2 Priority Levels - The priority level (mask bits) is contained in bits <7:5> of the PS and is used by software to determine which interrupts will be processed, as indicated in Table 1-2.

Table 1-2 Priority Levels

Octal Value of PS<7:5>	Interrupt Level Acknowledged
7	None
6	7
5	7, 6
4	7, 6, 5
3	7, 6, 5, 4
2	7, 6, 5, 4, 3
1	7, 6, 5, 4, 3, 2
0	7, 6, 5, 4, 3, 2, 1

1.3.3 The Trace/Trap Bit - The trace/trap bit (bit 4) is used for program debugging, enabling single-step execution of instructions for step-by-step monitoring.

1.3.4 Condition Codes - The four condition codes N, Z, V, and C contain information about the result of the last CPU operation. These bits are set as described in Paragraph 1.3.

1.3.5 Processor Status (PS) Protection - Tables 1-3, 1-4, 1-5, 1-6, and 1-7 summarize how the PS is protected under a variety of conditions. The PS is initialized at power-up (the value to which it is initialized depends on power-up options) and is cleared at console start. The RESET instruction does not affect the PS.

Table 1-3 PS Protection For Explicit Accesses

PS Bit(s)	EXPLICIT PS ACCESS		
	User	Super	Kernel
Condition Codes PS <3:0>	loaded from source	loaded from source	loaded from source
Trap Bit PS <4>	un-changed	un-changed	un-changed
Processor Priority PS <7:5>	loaded from source	loaded from source	loaded from source
Register Select PS <11>	loaded from source	loaded from source	loaded from source
Previous Mode PS <13:12>	loaded from source	loaded from source	loaded from source
Current Mode PS <15:14>	loaded from source	loaded from source	loaded from source

Table 1-4 PS Protection For Traps and Interrupts

PS Bit(s)	TRAPS & INTERRUPTS User	Super	Kernel
Condition Codes PS <3:0>	loaded from vector	loaded from vector	loaded from vector
Trap Bit PS <4>	loaded from vector	loaded from vector	loaded from vector
Processor Priority PS <7:5>	loaded from vector	loaded from vector	loaded from vector
Register Select PS <11>	loaded from vector	loaded from vector	loaded from vector
Previous Mode PS <13:12>	copied from PS <15:14>	copied from PS <15:14>	copied from PS <15:14>
Current Mode PS <15:14>	loaded from vector	loaded from vector	loaded from vector

Table 1-5 PS Protection For RTI, RTT Instructions

PS Bit(s)	RTI, RTT User	RTI, RTT Super	RTI, RTT Kernel
Condition Codes PS <3:0>	loaded from stack	loaded from stack	loaded from stack
Trap Bit PS <4>	loaded from stack	loaded from stack	loaded from stack
Processor Priority PS <7:5>	un-changed	un-changed	loaded from stack
Register Select PS <11>	ORed from stack*	ORed from stack*	loaded from stack
Previous Mode PS <13:12>	ORed from stack*	ORed from stack*	loaded from stack
Current Mode PS <15:14>	ORed from stack*	ORed from stack*	loaded from stack

* "ORed from stack" means that when the old PS is popped from the stack (restored), it cannot clear PS<15:11> in the current PS if these bits have been set.

Table 1-6 PS Protection for MTPS Instruction

PS Bit(s)	MTPS User	MTPS Super	MTPS Kernel
Condition Codes PS <3:0>	loaded from source	loaded from source	loaded from source
Trap Bit PS <4>	unchanged	unchanged	unchanged
Processor Priority PS <7:5>	unchanged	unchanged	loaded from source
Register Select PS <11>	unchanged	unchanged	unchanged
Previous Mode PS <13:12>	unchanged	unchanged	unchanged
Current Mode PS <15:14>	unchanged	unchanged	unchanged

Table 1-7 PS Initialization During Power-Up

PS Bit(s)	POWER-UP
Condition Codes PS <3:0>	cleared
Trap Bit PS <4>	cleared
Processor Priority PS <7:5>	depends on power-up option
Register Select PS <11>	cleared
Previous Mode PS <13:12>	cleared
Current Mode PS <15:14>	cleared i.e., kernel mode

1.4 INTERRUPTS AND TRAPS

This paragraph provides a brief overview of DCJ11 interrupts and traps and describes user-visible registers related to interrupts and traps. Abort conditions are also covered. For detailed timing and bus information, see Chapter 3 - Bus Cycles.

Interrupts and traps are requests that cause the DCJ11 to temporarily suspend the execution of the current program and provide service for the device or condition that caused the interrupt or trap. Interrupts differ from traps in that interrupts are initiated by some external event, while traps are caused by conditions internal to the DCJ11.

The DCJ11 operates at any of 8 levels of priority. In general, an interrupt or trap affects the DCJ11 if its priority is greater than the DCJ11's priority as indicated by PS<7:5>. The exception to this is a non-maskable interrupt or trap, which occurs independently of the processor priority. Note that non-maskable

interrupts and traps have a priority structure amongst themselves.

When an interrupt or trap occurs, the current PS and PC are preserved in order to allow a return to the interrupted program. The new contents of the PC and the PS are fetched from two consecutive memory words called a vector. The first word of the vector contains the interrupt or trap service routine starting address (the new PC), and the second word contains the new PS. Vectors are either predefined by the DCJ11 or are user defined. User defined vectors are vectors associated with interrupts occuring on IRQ<3:0>. The predefined vectors are shown in Table 1-8.

Specifically, for an interrupt or trap, the following sequence of events occurs:

```
          PS --> temp1      ;save PS, PC in temporary
          PC --> temp2      ;scratchpad locations
           0 --> PS<15:14>  ;force kernel mode
        M[V] --> PC         ;fetch PC from vector, data space
      M[V+2] --> PS         ;fetch PS from vector, data space
temp1<15:14> --> PS<13:12>  ;set previous mode
        SP-2 --> SP         ;pushed stack selected by new PS
       temp1 --> M[SP]      ;push old PS on stack, data space
        SP-2 --> SP
       temp2 --> M[SP]      ;push old PC on stack, data space
                            ;then execute interrupt service
                            ;routine
```

After the interrupt or trap service routine has been completed, an RTI (Return From Interrupt) or RTT (Return From Trap) instruction is typically executed. The top two words of the stack are automatically popped off the stack and placed in the PC and PS, respectively, thereby restoring the state of the interrupted program.

The DCJ11 also responds to a variety of conditions which can abort the current operation. An abort is similar to an interrupt or trap in that a vector is used to point to a service routine. Aborts differ from traps and interrupts in that the DCJ11 services an abort immediately rather than waiting until the end of the current macroinstruction. Aborts generated by the DCJ11 itself include memory management and address errors. Aborts which must be generated by external logic include bus timeouts, non-existent memory accesses, and parity aborts. The signal $\overline{\text{ABORT}}$ is asserted to indicate the presence of an abort condition.

DCJ11 interrupts, traps, and aborts (with their associated priorities) are summarized in Table 1-8. For interrupts and aborts, the name of the signal which initiates the interrupt or abort (if any) appears in the last column. For completeness, Table 1-8 also lists several instructions that result in traps. These instructions are mutually exclusive and have no priority structure.

Table 1-8 Interrupts, Traps, and Aborts

Description	Interrupt, Trap, or Abort	Vector Address	Priority Level	Signal
Red stack violation (CPU error register, bit 2)	Abort	4	NM	--
Address error (CPU error register, bit 6)	Abort	4	NM	--
Memory management violation (MMR0, bits <15:13>)	Abort	250	NM	--
Timeout/non-existent memory (CPU error register, bits <5:4>)	Abort	4	NM	$\overline{\text{ABORT}}$
Parity error	Interrupt or Abort	114	NM	$\overline{\text{PARITY}}$, $\overline{\text{ABORT}}$
Trace (T bit) set (PSW, bit 4)	Trap	14	NM	--
Yellow stack violation (CPU error register, bit 3)	Trap	4	NM	--
Power fail (PWRF)	Interrupt	24	NM	$\overline{\text{PWRF}}$
Floating point exception (FPA present)	Interrupt	244	NM	$\overline{\text{FPE}}$
Floating point exception (no FPA)	Trap	244	NM	--
PIR 7 (PIRQ, bit 15)	Trap	240	7	--
Interrupt level 7	Interrupt	UD	7	IRQ7
EVENT	Interrupt	100	6	$\overline{\text{EVENT}}$
PIR 6 (PIRQ, bit 14)	Trap	240	6	--
Interrupt level 6	Interrupt	UD	6	IRQ6
PIR 5 (PIRQ, bit 13)	Trap	240	5	--

Interrupt level 5	Interrupt	UD	5	IRQ5
PIR 4 (PIRQ, bit 12)	Trap	240	4	--
Interrupt level 4	Interrupt	UD	4	IRQ4
PIR 3 (PIRQ, bit 11)	Trap	240	3	--
PIR 2 (PIRQ, bit 10)	Trap	240	2	--
PIR 1 (PIRQ, bit 9)	Trap	240	1	--
TRAP Instruction	Trap	34	--	--
EMT Instruction	Trap	30	--	--
IOT Instruction	Trap	20	--	--
Illegal Instruction	Trap	10	--	--

NM = Non-maskable
UD = User-defined
-- = None

1.5 HALTING DCJ11 OPERATION

A halt operation differs from a interrupt, trap, or abort in that there is no vector associated with it. It is similar, however, in the sense that it interrupts the usual operation of the DCJ11. The two main means of halting the operation of the DCJ11 are to: (1) assert the HALT line or (2) execute a HALT instruction.

The HALT line has a lower priority than any interrupt, trap, or abort. However, it has the highest priority during vector reads. This is to allow the user to break out of potential infinite loops. An infinite loop could occur for example if a vector is not properly mapped during a memory management operation.

Execution of the HALT instruction performs different operations depending upon the CPU operating mode and the halt option currently selected. See Chapter 8 - Interfacing for more details on halt options. In kernel mode, a halt option of 1 causes a trap through location 4 and sets bit 7 of the CPU error register when HALT is executed. If the halt option is 0 in kernel mode, execution of the HALT instruction causes the DCJ11 into console ODT. Execution of the HALT instruction in user or supervisor mode causes a trap through location 4 and sets bit 7 of the CPU error register.

1.6 PROGRAM INTERRUPT REQUEST REGISTER

The program interrupt request register (PIRQ) provides seven levels of software interrupt (i.e., trap) capability. An interrupt request is queued by setting one of bits <15:9>, which correspond to interrupt priority levels 7 through 1 (respectively). Bits <7:5> and <3:1> are set by the DCJ11 to the encoded value of the highest pending request. When the program interrupt request is granted, the processor traps through a vector at virtual location 240. It is the responsibility of the interrupt service routine to clear the appropriate bit in the PIRQ before exiting. The format of the PIRQ is as shown in figure 1-4.

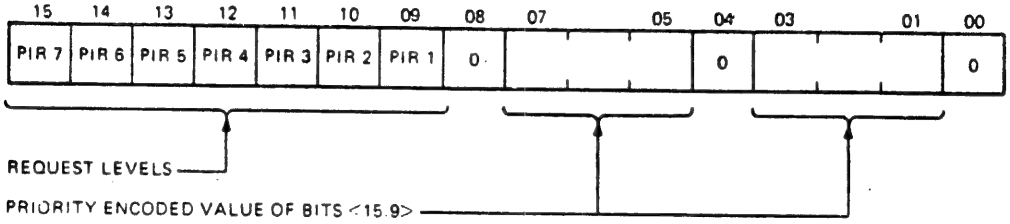


Figure 1-4 PIRQ Register

Bits <15:9> can be read or written. Bits <7:5> and <3:1> are read-only. The remaining bits are always read as zeros. PIRQ is cleared by a console start, by a RESET instruction, and at power-up time. The PIRQ resides at physical address 17777772.

1.7 CPU ERROR REGISTER

The CPU error register assists the operating system by identifying the source of a trap through location 4. The CPU error register is located at physical address 17777766. The format of the CPU error register is as shown in Figure 1-5.

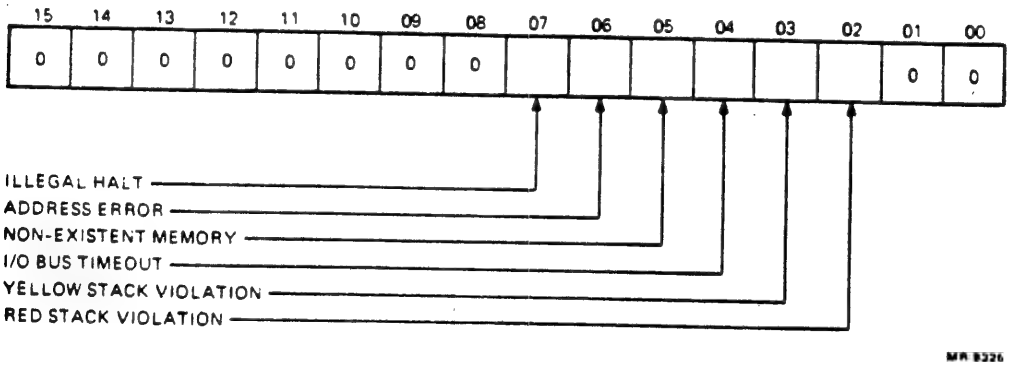


Figure 1-5 CPU Error Register

Bit	Name	Description
<15:8>	Unused	These bits are unused and are always read as zeros.
7	Illegal HALT (Read only)	Set when execution of a HALT instruction is attempted in user or supervisor mode, or in kernel mode when the HALT option is enabled (refer to the power-up options in Paragraph 8.3.3).
6	Address Error (Read only)	Set when a word access is made to an odd byte address, or when an instruction fetch from an internal register is attempted.
5	Non-Existent Memory (Read only)	Set when reference is made to a non-existent memory address.
4	I/O Bus Timeout (Read only)	Set when reference is made to a non-existent I/O page address.
3	Yellow Stack Trap (Read only)	Set when a yellow zone stack overflow trap occurs.
2	Red Stack Trap (Read only)	Set when a red stack trap occurs.
<1:0>	Unused	These bits are unused and are always read as zeros.

The CPU error register is cleared by any write reference to itself, by a power-up, or by a console start. The RESET instruction has no effect on this register.

1.8 STACK PROTECTION

The DCJ11 provides hardware protection for the kernel stack. The supervisor and user stacks are not protected by hardware but may be checked by memory management and appropriate software.

Stack protection in kernel mode is provided by defining yellow and red stack traps. Kernel stack references are checked against a fixed limit of 400 (octal). If the virtual address of a kernel stack reference is less than 400 (octal), a yellow stack trap occurs at the end of the current instruction. A stack trap can only occur on a kernel stack reference, which is defined as: any trap or interrupt push on the kernel stack, a JSR instruction in kernel mode, or a reference in kernel mode using addressing Mode 4

or 5 with R6 as the selected register.

The DCJ11 also checks for kernel stack aborts during interrupt, trap, or abort sequences. If an abort is caused by a kernel stack push during an interrupt, a trap, or an abort sequence, the DCJ11 initiates a red stack trap by creating an emergency stack at vector locations 0 and 2, vectoring through location 4, and setting bit 2 of the CPU error register.

1.9 FLOATING-POINT PROCESSING

The DCJ11 contains an integral floating-point processor which can perform single- and double-precision floating-point operations. User-accessible architecture associated with floating-point processing includes: six 64-bit floating-point accumulators (AC0--AC5), a floating-point status register (FPS), a floating-point exception address (FEA) register, and a floating-point exception code (FEC) register. Chapter 7 describes these in detail and provides information on programming with floating-point instructions.

1.10 MEMORY SYSTEM REGISTERS

Memory system registers are used for: (1) cache memory implementation and (2) memory management.

The memory system registers associated with cache memory are the cache control register (CCR) and the hit/miss register (HMR). These registers are described in detail in Chapter 5 - Special Features.

The memory system registers associated with memory management include page address registers (PARs), page descriptor registers (PDRs), and memory management registers 0, 1, 2, and 3 (MMR0, MMR1, MMR2, MMR3). These are described in detail in Chapter 4 - Memory Management.

1.11 DIRECT-MEMORY ACCESS (DMA) MECHANISM

An external device typically performs a DMA transfer by taking control of a buffered version of the DCJ11's data/address bus (DAL<21:00>). A device requests control of the DAL lines by asserting the $\overline{\text{DMR}}$ input to the DCJ11. This causes the DCJ11 to place DAL<15:00> in a high impedance state (DAL<21:16> is placed in a high impedance state via external buffers) and extend the current microcycle. It is the responsibility of external logic to end the microcycle by asserting the DCJ11's $\overline{\text{CONT}}$ input.

The DCJ11 acknowledges a DMA request by asserting its $\overline{\text{MAP}}$ output

at the appropriate time. See Chapter 3 - Bus Cycles for the specific timing involved. This also causes the current microcycle to extend until $\overline{\text{CONT}}$ is asserted.

A DMA request may be acknowledged and granted for all types of microcycles except bus writes and GP writes. The lack of a DMA grant, however, does not necessarily prevent external logic from performing a DMA transfer during these cycles. A buffered version of the DAL for example could be used for a DMA transfer when $\overline{\text{SCTL}}$ is asserted (the DAL itself would not be used since it carries the write data during this portion of the cycle).

NOTE

It is possible to acknowledge a DMA request between the read and write portions of a bus locked Read-Modify-Write cycle (see Paragraph 3.2). If this is not desirable, external logic should be designed to disable DMA requests at this time.

2.1 INTRODUCTION

This chapter describes the functions performed by each DCJ11 pin. The pins, and thus the chapter, are divided into nine groups:

- Data/address lines (DAL<21:00>)
- System control lines (BS<1:0>, AIO<3:0>, $\overline{\text{BUFCTL}}$, $\overline{\text{CONT}}$, DV)
- Timing signals ($\overline{\text{ALE}}$, $\overline{\text{SCTL}}$, $\overline{\text{STRB}}$, CLK, CLK2)
- Start/stop control ($\overline{\text{INIT}}$, $\overline{\text{HALT}}$)
- Status signals ($\overline{\text{MISS}}$, $\overline{\text{PARITY}}$, $\overline{\text{ABORT}}$, $\overline{\text{MAP}}$, $\overline{\text{PRDC}}$)
- Interrupt and DMA control (IRQ<3:0>, $\overline{\text{DMR}}$, $\overline{\text{PWRF}}$, $\overline{\text{FPE}}$, $\overline{\text{EVENT}}$)
- Test pins ($\overline{\text{TEST1}}$, $\overline{\text{TEST2}}$)
- Oscillator pins (XTLI, XTLO)
- Power pins (Vcc, GND)

Figure 2-1 illustrates the pin assignments of the DCJ11 and indicates whether a signal associated with a pin is an input, an output, or both (bidirectional).

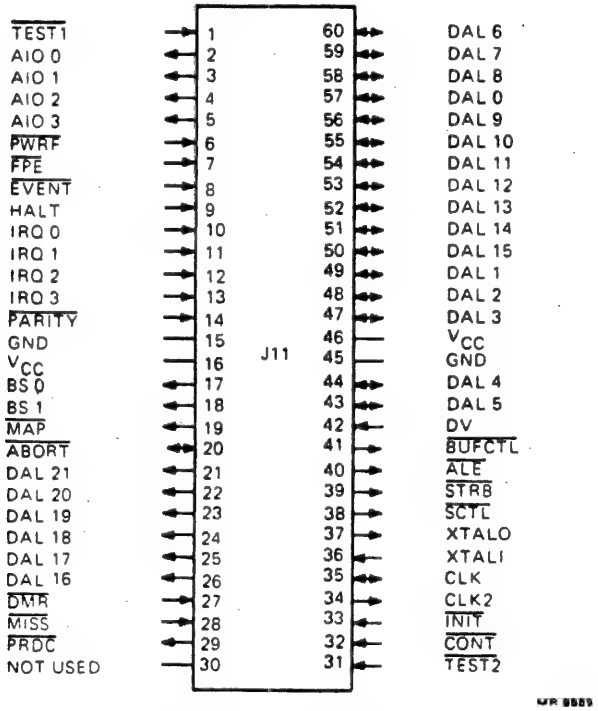


Figure 2-1 DCJ11 Pin Assignments

2.2 DATA/ADDRESS LINES (DAL<21:00>)

There are 22 pins associated with data and address information. These are usually referred to as the data/address (or DAL) lines. The DAL lines are functionally divided into two groups: the upper data/address lines (DAL<21:16>) which are output only and the lower data/address lines (DAL<15:00>) which are bidirectional.

2.2.1 Upper Data/Address Lines (DAL<21:16>) - These six time-multiplexed output lines constitute the most significant 6 bits of a 22-bit physical address. DAL<21:16> carries valid information at the beginning of every bus cycle. Internal status is asserted on these lines during the second part of every bus cycle for manufacturing test purposes only.

2.2.2 Lower Data/Address Lines (DAL<15:00>) - These time-multiplexed I/O lines constitute the 16-bit data and address bus. During the first part of a cycle that involves an I/O transfer, the DAL lines carry a physical address, an interrupt acknowledge priority level, or a general-purpose (GP) code, depending upon the type of cycle being performed (see Chapter 3 - Bus Cycles for more information on cycle types). During a Bus Read or Bus Write cycle, DAL<15:00> carries the lower 16 bits of a physical address. During an Interrupt Acknowledge cycle, DAL<3:0> carries the priority of the acknowledged level. During a General-Purpose Read or General-Purpose Write cycle, DAL<7:0> carries the GP code.

During the second part of a cycle that involves an I/O transfer, the DAL lines carry 8 or 16 bits of data. During read cycles, external logic places data onto the DAL. If the DCJ11 only requires a byte of information, it reads a full word but ignores either the upper of lower byte. For write cycles, the DAL carries 8 or 16 bits of data, depending upon whether the cycle involves the writing of a byte or a word.

2.3 SYSTEM CONTROL LINES.

There are nine pins associated with system control: BS<1:0>, AIO<3:0>, $\overline{\text{BUFCTL}}$, $\overline{\text{CONT}}$, and DV.

2.3.1 Bank Select (BS<1:0>) - These time-multiplexed output signals transmit bank select and cache access information. At the beginning of a Bus Read or Bus Write cycle, the BS signals define the type of device being accessed by the physical address on the DAL as shown in Table 2-1.

Table 2-1 BS Device Selection

BS1	BS0	DESCRIPTION
1	1	Internal register - A memory-addressable register that resides within the DCJ11. Included are the processor status word, all MMU registers, the PIRQ register, the CPU error register and the cache hit/miss register. Excluded are the general-purpose registers, which are not memory addressable.
1	0	External I/O device - Any device or register external to the DCJ11 that is referenced by a bus address in the upper 8K bytes of the physical address range (17760000 to 17777777). Excluded are system registers (BS code 01) and internal registers (BS code 11).
0	1	System register - A memory-addressable register in the address range 17777740 to 17777750. Always included as a system register is the DCJ11's internal cache control register (CCR). NOTE The CCR is the only system register implemented in the DCJ11. Accesses to the CCR generate the same BS code as for the other system registers mentioned above. This facilitates the creation of "shadow" read-only copies of the CCR on cache based systems.
0	0	Memory - A reference to any location in physical address space in the range 00000000 to 17757777.

During the second part of an I/O cycle, BS1 is asserted when the cache memory (if present) is to be bypassed. In the second part of the cycle, BS0 is asserted whenever a cache memory force miss is required.

2.3.2 Address Input/Output (AIO<3:0>) - The AIO outputs identify the type of cycle currently being executed. External logic typically latches and decodes these signals. Table 2-2 specifies the AIO code associated with each cycle type. See Chapter 3 - Bus Cycles for detailed information on the various cycle types.

Table 2-2 AIO Decode

AIO3	AIO2	AIO1	AIO0	CYCLE TYPE
1	1	1	1	NIO (internal operation only, no I/O)
1	1	1	0	GP (General-Purpose) read
1	1	0	1	Interrupt acknowledge, vector read
1	1	0	0	Instruction-stream request read
1	0	1	1	Read/Modify/Write - no bus lock
1	0	1	0	Read/Modify/Write - bus lock
1	0	0	1	Data-stream read
1	0	0	0	Instruction-stream demand read
0	1	0	1	GP word write
0	0	1	1	Bus byte write
0	0	0	1	Bus word write

2.3.3 Buffer Control ($\overline{\text{BUFCTL}}$) - The $\overline{\text{BUFCTL}}$ output defines whether the DCJ11 is driving or receiving data on the DAL. $\overline{\text{BUFCTL}}$ is typically used by external logic to control the direction of data passing through buffers that send data to the DCJ11. When asserted, $\overline{\text{BUFCTL}}$ indicates that the DCJ11 is not driving data on the DAL. This occurs: (1) during the portion of a read cycle when data is being driven on the DAL, and (2) during the stretched portion of any nonwrite cycle. $\overline{\text{BUFCTL}}$ is deasserted when the DCJ11 is driving data or an address on the DAL.

2.3.4 Continue ($\overline{\text{CONT}}$) - The $\overline{\text{CONT}}$ input is asserted by external logic to terminate a stretched cycle after it has finished using the DAL. $\overline{\text{CONT}}$ is so named because it enables the DCJ11 to continue on to the next cycle.

2.3.5 Data Valid (DV) - The DV input is typically asserted by external logic to latch data into the DCJ11 from the DAL. When asserted, DV causes the DCJ11 to latch data when BUFCTL and SCTL are asserted, that is, during stretched non-write cycles. External logic must ensure that DV is not asserted during DMA transactions, since this would cause the latching of unpredictable data.

2.4 TIMING SIGNALS

There are five pins associated with timing and synchronization: $\overline{\text{ALE}}$, $\overline{\text{SCTL}}$, $\overline{\text{STRB}}$, CLK, and CLK2.

2.4.1 Address Latch Enable ($\overline{ALE}$) - $\overline{ALE}$ when asserted indicates that DAL<21:00>, AIO<3:0>, BS<1:0>, and $\overline{MAP}$ all contain valid data. The leading edge of ALE is typically used by external logic to latch addresses, AIO codes, bank select (BS) codes, and the map enable ($\overline{MAP}$) control signal.

2.4.2 Stretch Control ($\overline{SCTL}$) - The $\overline{SCTL}$ output, when asserted, identifies the stretched portion of a cycle. During write cycles, the leading or trailing edge of $\overline{SCTL}$ can be used for latching data. During read cycles, the trailing edge of $\overline{SCTL}$ can be used for latching data. $\overline{SCTL}$ can also be used to determine when externally generated aborts may occur.

2.4.3 Strobe ($\overline{STRB}$) - The assertion of the $\overline{STRB}$ output occurs one clock period after the assertion of $\overline{ALE}$. The deassertion of $\overline{STRB}$ identifies the end of one microcycle and the beginning of another. $\overline{STRB}$ is a general-purpose strobe signal and is typically used for system bus control.

2.4.4 Clock 1 (CLK) - CLK is usually a clock output for diagnostic use only. When used as an output, CLK reflects the state of the DCJ11's internal clock. The frequency of CLK equals the frequency of the external crystal oscillator circuit connected to the XTALI and XTALO pins. If $\overline{TEST2}$ is asserted, the DCJ11's internal clock is disabled and CLK is placed in the high-impedance state. In this case, CLK can serve as a MOS input (V_{TL} = .3Vcc, V_{TH} = .7Vcc, $t_{LH} = t_{HL}$ = 7 ns) driven by an external clock.

2.4.5 Clock 2 (CLK2) - The CLK2 output has the same frequency as CLK. Like CLK, CLK2 reflects the state of the DCJ11's internal clock and is disabled by the assertion of $\overline{TEST2}$. Unlike CLK, CLK2 is typically used as a system clock or master clock for external logic. CLK and CLK2 have minimal skew when loaded equally.

2.5 START/STOP CONTROL

There are two pins associated with starting and stopping the operation of the DCJ11: $\overline{INIT}$ and HALT.

2.5.1 Initialize ($\overline{INIT}$) - The $\overline{INIT}$ input, when asserted, initializes (resets) the DCJ11 by forcing it through a power-up procedure. The power-up sequence is described in detail in Paragraph 8.3.2.

2.5.2 Halt (HALT) - The HALT input, when asserted, forces the DCJ11 into console mode (i.e., initiates console ODT). HALT is the lowest priority nonmaskable interrupt except during vector read cycles. During vector read cycles, HALT becomes the highest priority non-maskable interrupt. This allows escape from potential infinite looping which could result from programming errors. Since it is non-maskable, HALT is unaffected by the CPU priority specified by PS<7:5>. See Chapter 1 - Architecture for a list of the non-maskable interrupts and their relative priorities. See Chapter 5 - Special Features for a description of console ODT.

2.6 STATUS SIGNALS

There are five pins associated with indicating DCJ11 status: $\overline{\text{MISS}}$, $\overline{\text{PARITY}}$, $\overline{\text{ABORT}}$, $\overline{\text{MAP}}$, and $\overline{\text{PRDC}}$.

2.6.1 Cache Miss ($\overline{\text{MISS}}$) - The $\overline{\text{MISS}}$ input is generated by external logic in DCJ11 based systems incorporating cache memory. The assertion of $\overline{\text{MISS}}$ typically indicates that the current memory reference resulted in a cache memory miss. If $\overline{\text{MISS}}$ is asserted during the first part of a bus read cycle, the cycle is stretched.

2.6.2 Parity Error ($\overline{\text{PARITY}}$) - The assertion of the $\overline{\text{PARITY}}$ input indicates the occurrence of a memory parity error. $\overline{\text{PARITY}}$ is used to generate parity aborts and parity interrupts. If $\overline{\text{PARITY}}$ is asserted and $\overline{\text{ABORT}}$ is also asserted, then a parity error abort is generated. The DCJ11 immediately traps through a vector located at virtual address 114 without completing the current instruction. If $\overline{\text{PARITY}}$ is asserted but $\overline{\text{ABORT}}$ is not asserted, then a parity error interrupt is generated. At the end of the current instruction, the interrupt is serviced through the vector located at virtual address 114. Note that $\overline{\text{PARITY}}$ is sampled only during the stretched portion of a cycle.

2.6.3 Abort ($\overline{\text{ABORT}}$) - $\overline{\text{ABORT}}$ can serve as an input or an output of the DCJ11. $\overline{\text{ABORT}}$ is typically configured in an open-collector driver circuit such that aborts generated by either external logic or the DCJ11 can cause $\overline{\text{ABORT}}$ to be asserted (i.e., a wired OR arrangement). Note that the DCJ11 pulls $\overline{\text{ABORT}}$ high internally.

The DCJ11 asserts $\overline{\text{ABORT}}$ during the first part of an I/O cycle if a memory management error or address error occurs. For a memory management error, the DCJ11 traps through a vector located at virtual address 250 in kernel data space. For an address error, the DCJ11 traps through a vector located at virtual address 4 in kernel data space. The DCJ11 sets the appropriate bit in the CPU error register.

$\overline{\text{ABORT}}$ can also be asserted by external logic in the event of such conditions as a bus timeout, non-existent memory reference, parity

error, etc. External logic must ensure that: (1) the cycle is stretched and that $\overline{ABORT}$ is asserted during the stretched portion (i.e., when $\overline{SCTL}$ is asserted) and (2) $\overline{ABORT}$ is not asserted during a non-I/O cycle. If $\overline{PARITY}$ is not asserted, the assertion of $\overline{ABORT}$ by external logic causes a trap through a vector located at virtual address 4 in kernel data space. The CPU error register specifies the cause of the abort. If $\overline{PARITY}$ and $\overline{ABORT}$ are asserted, the DCJ11 immediately performs a trap through a vector located at virtual address 114 in virtual address space.

2.6.4 Map Enable ($\overline{MAP}$) - $\overline{MAP}$ is a time-multiplexed output. The assertion of $\overline{MAP}$ during the first part of a cycle indicates that the I/O map has been enabled (the I/O map is enabled by setting bit 5 of MMR3 to 1). The assertion of $\overline{MAP}$ during the second part of a cycle acknowledges the assertion of the DMR input.

NOTE
The I/O map, if needed, is implemented in circuitry external to the DCJ11.

2.6.5 Predecode ($\overline{PRDC}$) - The $\overline{PRDC}$ output, when asserted, indicates that the contents of the prefetch buffer (PB) are being decoded as the next macroinstruction. This implies that the contents of the PB are valid. The PB is part of the DCJ11 prefetch pipeline, the operation of which is explained in Chapter 5 - Special Features.

2.7 INTERRUPT AND DMA CONTROL

There are eight pins associated with the control of program interrupts and DMA transfers: IRQ<3:0>, $\overline{DMR}$, $\overline{PWRF}$, $\overline{FPE}$, and $\overline{EVENT}$.

2.7.1 Interrupt Request (IRQ<3:0>) - IRQ<3:0> are four input lines that correspond to four different levels of external interrupt requests. Interrupt requests at any of these four levels can be masked by PS<7:5>. In order to be serviced, the requesting device must have an interrupt priority higher than the priority indicated by PS<7:5>. Interrupt requests on IRQ<3:0> are blocked or allowed as summarized in Table 2-3:

Table 2-3 Interrrupt Requests on IRQ<3:0>

PS<7:5>	CPU Priority Level	IRQ3	IRQ2	IRQ1	IRQ0
111	7	Blocked	Blocked	Blocked	Blocked
110	6	Allowed	Blocked	Blocked	Blocked
101	5	Allowed	Allowed	Blocked	Blocked
100	4	Allowed	Allowed	Allowed	Blocked
0xx	3-0	Allowed	Allowed	Allowed	Allowed

x = Irrelevant

From Table 2-3, it is seen that each IRQ line is associated with a different interrupt level, as summarized in Table 2-4.

Table 2-4 IRQ<3:0> Interrupt Request Levels

IRQ Line	Interrupt Request Level
IRQ3	7
IRQ2	6
IRQ1	5
IRQ0	4

2.7.2 Direct Memory Access Request ($\overline{DMR}$) - The $\overline{DMR}$ input to the DCJ11 when asserted typically means that an external device wants to perform a DMA transaction. $\overline{DMR}$ is sampled by the DCJ11 at the start of all cycles. If the cycle does not involve a write operation, the DCJ11 responds to the assertion of $\overline{DMR}$ by: (1) stretching the cycle, (2) placing DAL<15:00> in the high-impedance state, and (3) acknowledging the DMA request by asserting $\overline{MAP}$ during the second part of the cycle. If the cycle involves a write operation, the cycle is stretched but DAL<15:00> is not placed in the high-impedance state and $\overline{MAP}$ is not asserted.

2.7.3 Power Fail ($\overline{PWRF}$) - $\overline{PWRF}$ is a high-priority nonmaskable interrupt input that, when asserted, forces a trap through a vector located at virtual address 24 in kernel data space. External logic typically asserts $\overline{PWRF}$ to indicate the occurrence of an AC power failure. The trap vector points to an appropriate user-defined power fail service routine.

2.7.4 Floating-Point Exception ($\overline{FPE}$) - $\overline{FPE}$ is a high-priority nonmaskable interrupt input that, when asserted, forces a trap through a vector located at virtual address 244 in kernel data space. $\overline{FPE}$ would be asserted by an external FPA coprocessor to indicate the occurrence of a floating-point exception. The trap vector would point to an appropriate user-defined floating-point exception service routine.

2.7.5 Event ($\overline{\text{EVENT}}$) - The $\overline{\text{EVENT}}$ input is a maskable priority level 6 interrupt (i.e., it is acknowledged if PS<7:5> is less than 6). When $\overline{\text{EVENT}}$ is asserted (and not masked), the DCJ11 performs a trap through a vector located at virtual address 100 in kernel data space. $\overline{\text{EVENT}}$ is typically used by external logic as a line time clock (LTC) interrupt input.

2.8 TEST PINS

There are two pins associated with testing, $\overline{\text{TEST1}}$ and $\overline{\text{TEST2}}$. These signals disable DCJ11 functions and are are used in connection with board-level testing.

2.8.1 Test 1 ($\overline{\text{TEST1}}$) - The $\overline{\text{TEST1}}$ input (when asserted by external logic) disables all DCJ11 outputs by placing them in the high-impedance state. This permits external logic to operate on the data and control lines connected to the DCJ11 without interference from the DCJ11.

2.8.2 Test 2 ($\overline{\text{TEST2}}$) - The $\overline{\text{TEST2}}$ input, when asserted, disables the DCJ11's internal clock. The CLK and CLK2 pins are placed in the high-impedance state. Board level in-circuit testing logic can be designed such that when $\overline{\text{TEST2}}$ is asserted, an external clock drives the DCJ11 clock circuitry through the CLK pin.

2.9 OSCILLATOR PINS (XTALI, XTALO)

The XTALI and XTALO pins are used to connect an external crystal circuit to the DCJ11. The recommended crystal circuit is shown in Figure 2-2.

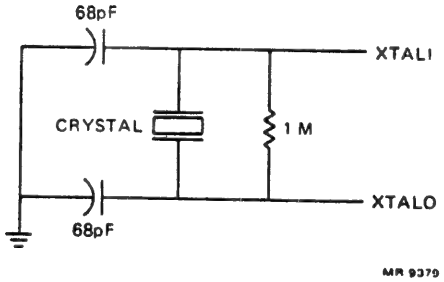


Figure 2-2 Typical XTALI and XTALO Generation

2.10 POWER PINS

There are four pins associated with power: two for +5VDC (Vcc) and two for ground (GND).

2.10.1 Power (Vcc) - There are two pins, both called Vcc, which are used to input +5VDC to the DCJ11. +5VDC is supplied by external circuitry and is typically maintained to within ± 5%.

2.10.2 Ground (GND) - The two GND pins provide a ground reference for the DCJ11. Typically, these pins are connected to the ground reference of external logic.

2.11 PIN DESCRIPTION SUMMARY

PIN NO.	PIN NAME	DEFINITION	INPUT OR OUTPUT	FUNCTION
1	$\overline{\text{TEST1}}$	Test 1	Input	Disables all DCJ11 outputs.
2-5	AIO<3:0>	Address Input/Output	Output	Indicate the type of cycle currently being executed (e.g., bus read, GP write, IACK, etc.)
6	$\overline{\text{PWRF}}$	Power Fail	Input	A high-priority non-maskable interrupt that forces a trap through vector location 24. Indicates an AC power failure.
7	$\overline{\text{FPE}}$	Floating-Point Exception	Input	A high-priority non-maskable interrupt that forces a trap through vector location 244. Typically generated by a floating-point coprocessor to indicate an exception condition.
8	$\overline{\text{EVENT}}$	Event	Input	A maskable interrupt that forces a trap through vector location 100. Typically used as a line time clock.

9	HALT	Halt	Input	A low-priority non-maskable interrupt that forces the DCJ11 into console ODT.
10-13	IRQ<3:0>	Interrupt Request	Input	Four maskable interrupt request lines.
14	$\overline{\text{PARITY}}$	Parity Error	Input	Indicates a memory parity error.
15	GND	Ground	Input	Ground reference.
16	Vcc	Power	Input	+5 VDC power input.
17-18	BS<1:0>	Bank Select	Output	Multiplexed. Either define the type of physical address on the DAL or indicate if a cache memory bypass or force miss should occur.
19	$\overline{\text{MAP}}$	Map Enable	Output	Multiplexed, indicates that either the I/O map is enabled or a DMA request has been granted.
20	$\overline{\text{ABORT}}$	Abort	I/O	Indicates the occurrence of an abort condition, i.e.; a memory management or address error, bus timeout, non-existent memory, or parity error.
21-26	DAL<21:16>	Data/Address Lines	Output	Most significant six bits of the time multiplexed data and address bus.
27	$\overline{\text{DMR}}$	Direct Memory Access Request	Input	Forces the current cycle to be extended and causes $\overline{\text{MAP}}$ to be asserted during the second part of the cycle.

28	$\overline{\text{MISS}}$	Cache Miss	Input	Indicates whether the current memory reference resulted in a cache hit or miss.
29	$\overline{\text{PRDC}}$	Predecode	Output	Indicates when the contents of the prefetch buffer are being decoded as the next macroinstruction.
30	Not Used			
31	$\overline{\text{TEST2}}$	Test 2	Input	Disables the clock outputs. Permits external logic to drive the DCJ11's internal clock circuitry through the CLK pin.
32	$\overline{\text{CONT}}$	Continue	Input	Terminates a stretched cycle.
33	$\overline{\text{INIT}}$	Initialize	Input	Initializes or resets the system by forcing it through a power-up procedure.
34	CLK2	Clock 2	Output	Clock output with the same frequency as CLK. Typically used as a system clock.
35	CLK	Clock 1	Output	Clock output for diagnostic use only.
36	XTALI	Crystal Input	Input	Oscillator input line.
37	XTALO	Crystal Output	Output	Oscillator output line.
38	$\overline{\text{SCTL}}$	Stretch Control	Output	Indicates that a cycle is being stretched. The edges can be used to strobe data.
39	$\overline{\text{STRB}}$	Strobe	Output	General-purpose strobe.
40	$\overline{\text{ALE}}$	Address Latch Enable	Output	Typically used to latch addresses, AIO codes, and the map enable and BS control signals.

41	$\overline{\text{BUFCTL}}$	Buffer Control	Output	Indicates the direction of data on the DAL. Asserted when the DCJ11 is not driving the DAL.
42	DV	Data Valid	Input	Causes the DCJ11 to to latch data from the DAL.
43-44, 47-60	DAL<15:00>	Data/Address Lines	I/O	Lower 16 bits of the time multiplexed data and address bus.
45	GND	Ground	Input	Ground reference.
46	Vcc	Power	Input	+5 VDC power input.

3.1 INTRODUCTION

This chapter describes the various types of DCJ11 bus cycles. A bus cycle is a sequence of events which defines the activity on the DCJ11's I/O bus. Bus cycles are also sometimes referred to as "microcycles", since each bus cycle is associated with the execution of one microinstruction. The execution of a DCJ11 macroinstruction such as ADD, JMP, etc., can involve the execution of several bus cycles. The type of bus cycle that the DCJ11 performs depends upon the type of bus activity (if any) required to complete the execution of a microinstruction.

Sometimes the DCJ11 performs an internal operation which requires no bus activity. If this is the case, the DCJ11 executes a non-I/O (NIO) cycle. An NIO bus cycle (described in detail in Paragraph 3.4) is the only type of bus cycle that does not involve the transfer of information over the DCJ11's I/O bus.

DCJ11 bus cycles fall into six broad categories:

1. Non-I/O
2. Bus Read
3. Bus Write
4. General-Purpose Read
5. General-Purpose Write
6. Interrupt Acknowledge

The deassertion of the signal $\overline{\text{STRB}}$ marks the beginning (and the end) of a bus cycle. $\overline{\text{ALE}}$ (asserted shortly after $\overline{\text{STRB}}$ is deasserted) can be used by external logic to latch AIO<3:0>. The information on AIO<3:0> specifies the type of bus cycle being performed according to Table 3-1:

Table 3-1 AIO Codes for Bus Cycles

AIO<3:0>	Description	Bus Cycle Type
1111	Non-I/O operation	Non-I/O
1110	GP read	General-Purpose Read
1101	Interrupt acknowledge/ vector read	Interrupt Acknowledge
1100	Instruction stream request read	Bus Read
1011	Read-Modify-Write, no bus lock	Bus Read*
1010	Read-Modify-Write, bus lock	Bus Read*
1001	Data stream read	Bus Read
1000	Instruction stream demand read	Bus Read
0101	GP word write	General-Purpose Write
0011	Bus byte write	Bus Write
0001	Bus word write	Bus Write

* Note that the AIO codes for read-modify-write cycles are identified as Bus Read cycles. This refers to the first part of the cycle (i.e., the "read" part). The second part of the cycle (i.e., the "write" part) will be a Bus Write cycle with a different AIO code.

3.2 DURATION OF BUS CYCLES

The length of a bus cycle is usually expressed as a number of periods of the DCJ11's master clock (CLK). All bus cycles last for a minimum of four clock periods. However, cycles may be extended or "stretched" beyond this minimum by an internal event or by external logic. When a cycle is stretched, it is always stretched for a minimum of four additional clock periods. A cycle can continue to be stretched in increments of two periods and can remain stretched indefinitely. Stretched cycles are ended by the assertion of the signal $\overline{\text{CONT}}$. $\overline{\text{CONT}}$ is sampled by the DCJ11 on the first falling edge of T4 and on every other succeeding falling edge of T4.

A bus cycle will be stretched unless either of the following two groups of conditions exists:

1. A Bus Read cycle is executed and BS<1:0> = 00 throughout the cycle (i.e., the cycle involves a memory read and does not involve a cache bypass or force miss) and $\overline{\text{DMR}}$ and $\overline{\text{MISS}}$ are not asserted during the cycle (no DMA grant or cache miss). Furthermore, $\overline{\text{ABORT}}$ must not be asserted if the cycle involves an instruction stream demand read.

2. A Non-I/O cycle is executed and $\overline{\text{DMR}}$ is not asserted during the cycle.

Timing diagrams for both stretched and non-stretched cycles are provided in the paragraphs that follow.

3.3 Bus Cycle Parts

Reference is sometimes made to the "first" (or "early") part and the "second" (or "later") part of a bus cycle. The first part of a bus cycle is defined as the duration of the first two clock periods, shown as T0 and T1 in the bus cycle timing diagrams. The second part of a bus cycle is defined as the duration of the remaining clock periods in the cycle. A non-stretched cycle has only two clock periods in its second part. These are shown as T2 and T3 in the bus cycle timing diagrams. A stretched cycle has at least six clock periods in its second part. These are shown as T2 through T7 in the bus cycle timing diagrams. Note that if a cycle is stretched for more than six clock periods in its second part, T4 is repeated in pairs.

3.4 NON-I/O (NIO) CYCLE

When the DCJ11 executes a microinstruction which involves no interaction with external logic (i.e., requires no I/O bus activity), it performs a Non-I/O (or NIO) cycle. Non-stretched and stretched Non-I/O cycles are illustrated in Figures 3-1 and 3-2, respectively.

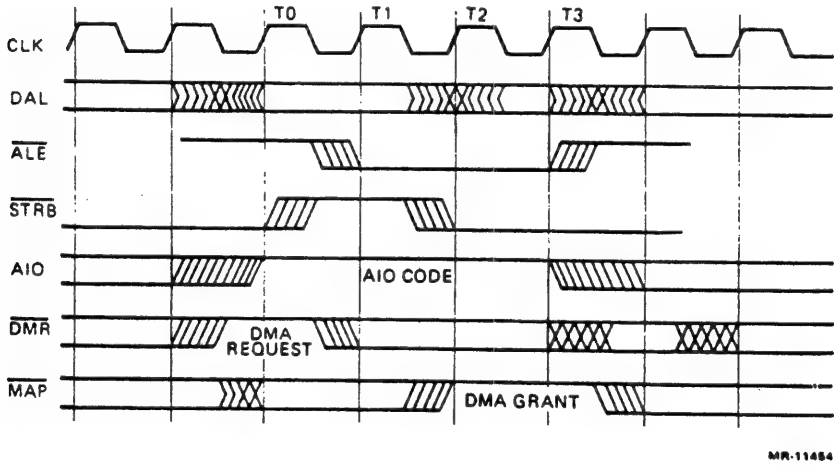


Figure 3-1 Non-Stretched Non-I/O Cycle

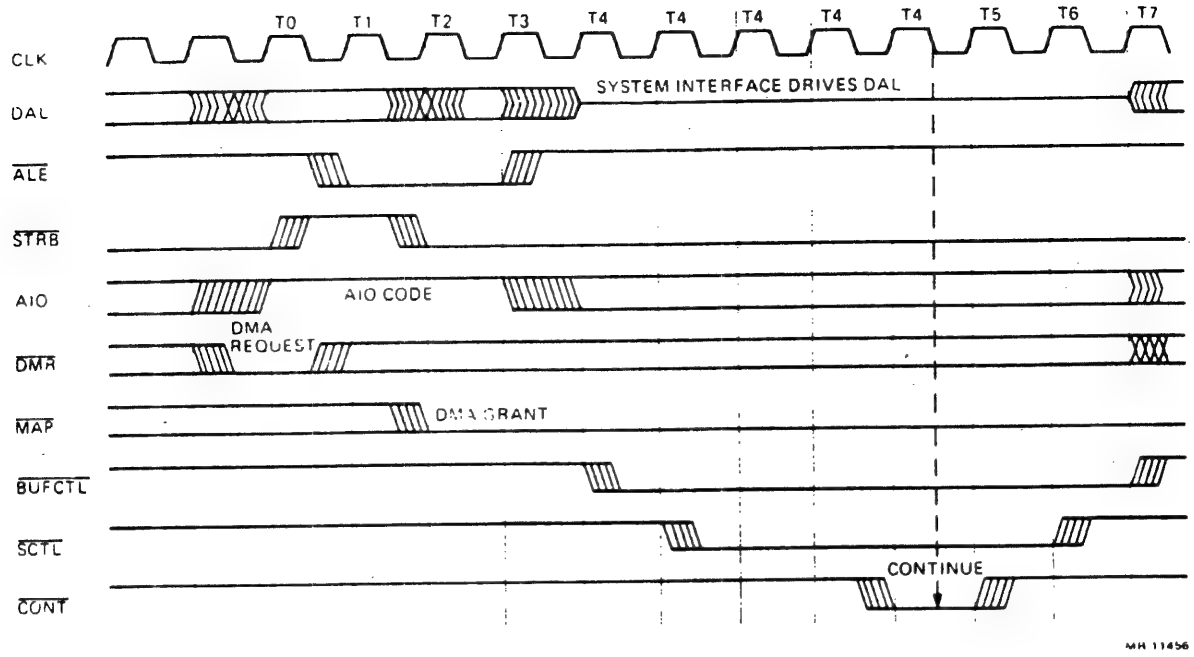


Figure 3-2 Stretched Non-I/O Cycle

The deassertion of STRB marks the beginning of the cycle, which is followed shortly afterwards by the assertion of ALE. ALE typically latches the AIO code which identifies the cycle as non-I/O. The DAL, BS<1:0>, MAP, and ABORT outputs are undefined and should be ignored by external logic. External logic must not assert ABORT during an NIO cycle. If a direct memory access request (DMR) is granted, the cycle is stretched and SCTL and BUFCTL are asserted.

As shown in Figure 3-1, a non-stretched NIO cycle is four clock periods in duration. If a DMA request is received during the first part of the cycle the cycle is stretched to eight or more clock periods (note the assertion of DMR during the first part of the cycle in Figure 3-2). Otherwise, the cycle does not stretch. If the NIO cycle is stretched, BUFCTL and SCTL are asserted during the stretched part of the cycle. The time-multiplexed signal MAP asserted during the second part of the stretched cycle indicates the granting of the DMA request. The cycle continues to be stretched in increments of two clock periods (T4) until CONT is asserted.

3.5 BUS READ CYCLE

The different types of bus read cycles which the DCJ11 can perform include instruction-stream request or demand reads, data-stream reads, and the read portion of a read/modify/write cycle. The AIO code defines which of these is selected. The types of devices from which information can be read include memory, I/O devices, and explicitly addressable registers. During the first part of the cycle, BS<1:0> defines which of these is selected. All read cycles involve the reading of a full word. If the DCJ11 needs only a byte, it reads a word and ignores the unused byte.

Note the distinction between request reads and demand reads. A request read occurs when the DCJ11 is prefetching information. If an abort occurs at this time, it does not affect macroinstruction flow (i.e., aborts are ignored). All other types of reads are demand reads, during which aborts are recognized and serviced via the service vectors shown in Table 1-8.

Non-stretched and stretched Bus Read cycles are illustrated in Figure 3-3 and 3-4, respectively.

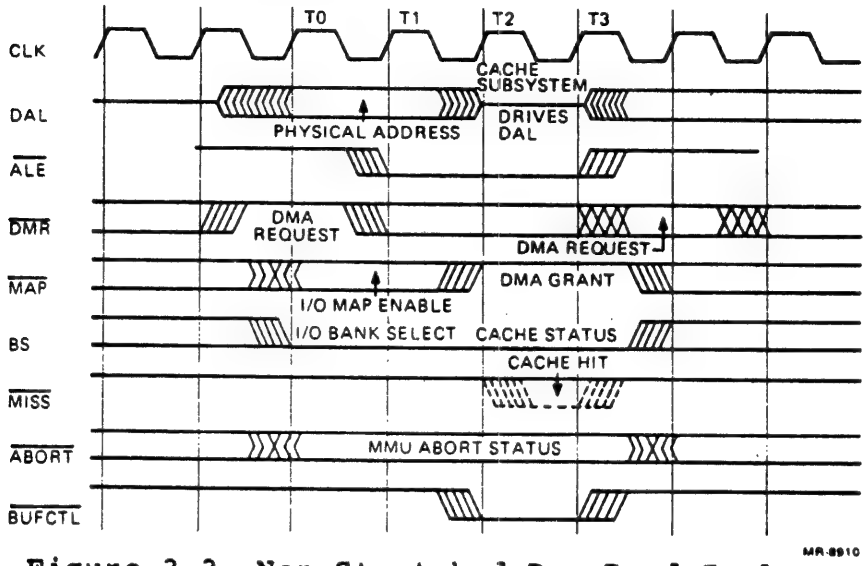


Figure 3-3 Non-Stretched Bus Read Cycle

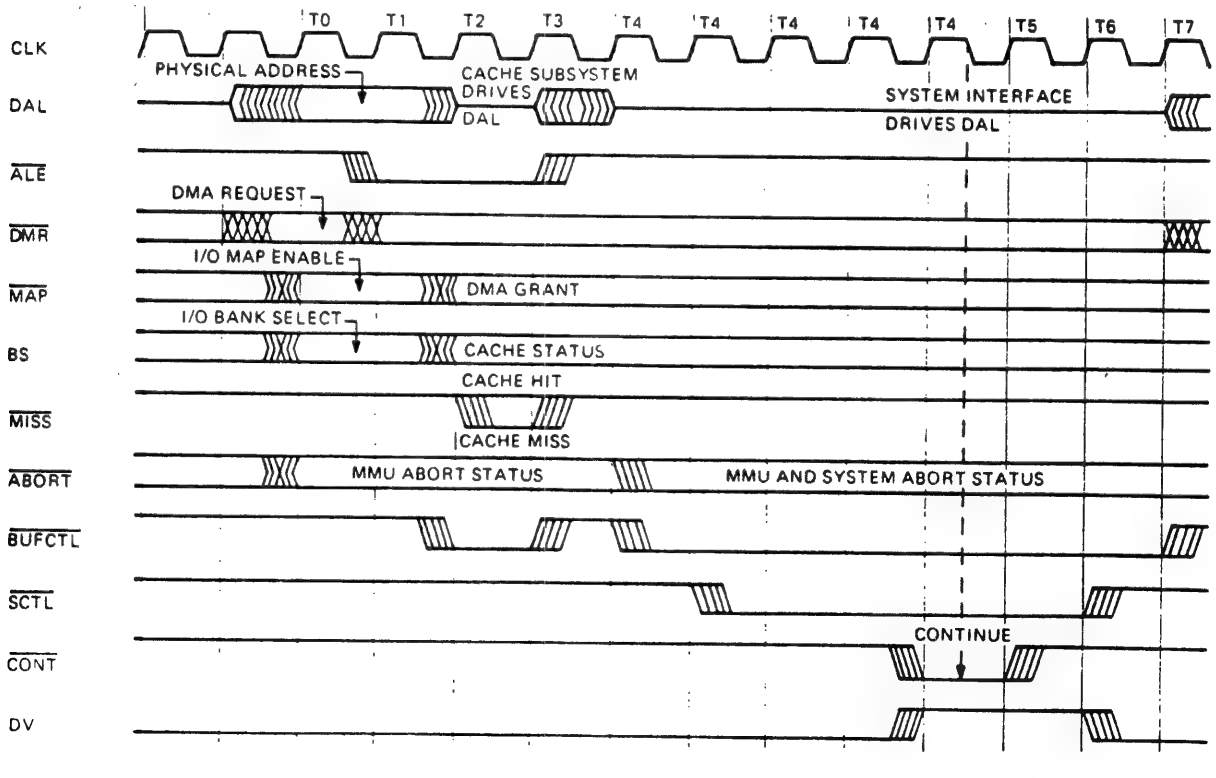


Figure 3-4 Stretched Bus Read Cycle

$\overline{ALE}$ can be used to latch the AIO code, the physical address on the data/address lines (DAL), the Bank Select (BS) information, and I/O Map Enable ($\overline{MAP}$) information.

A Bus Read cycle will stretch if any of the following conditions exist:

- BS<1:0> does not equal 00 during the first part of the cycle (anything other than a memory reference)
- BS<1:0> does not equal 00 during the second part of the cycle (a cache memory force miss or a cache bypass)
- $\overline{MAP}$ is asserted during the second part of the cycle (a DMA grant)
- $\overline{MISS}$ is asserted during the second part of the cycle (a cache miss)
- $\overline{ABORT}$ is asserted by the DCJ11 during an instruction stream demand read, data stream read, or read-modify-write cycle

Otherwise, a Bus Read cycle will execute in four clock periods.

For non-stretched Bus Read cycles, the read data is synchronously latched into the DCJ11 only on the rising edge of T3, as shown in Figure 3-3.

For stretched Bus Read cycles, data is latched into the DCJ11 both at the rising edge of T3 and when DV is asserted during the stretched portion of the cycle (see Figure 3-4). Thus if read data is valid at the rising edge of T3, it is latched at that time and DV is not required. If the read data is not valid at the rising edge of T3, DV is required to latch the valid data. Note that DV should be inhibited if the stretched Bus Read is due only to a DMA grant.

A stretched cycle lasts at least eight clock periods. A cycle is stretched in increments of two clock periods (T4) and is ended by the assertion of $\overline{CONT}$.

If an internally generated abort condition such as an MMU error or address error exists, the DCJ11 asserts $\overline{ABORT}$ during the first part of the cycle. If this type of abort occurs, the DAL, BS, and $\overline{MAP}$ information should be ignored for the remainder of the cycle. If an abort is externally generated (such as bus timeout, non-existent memory reference, etc.), it must occur during the stretched portion of the cycle.

3.6 BUS WRITE CYCLE

There are two different types of bus write cycles: Bus Word Write cycles and Bus Byte Write cycles. The AIO code defines which of these is selected. The types of devices to which information can be written include memory, I/O devices, and bus addressable registers. During the first part of the cycle, BS<1:0> defines

which of these is selected.

Bus Write cycle timing is illustrated in Figure 3-5. Note that Bus Write cycles are always stretched cycles.

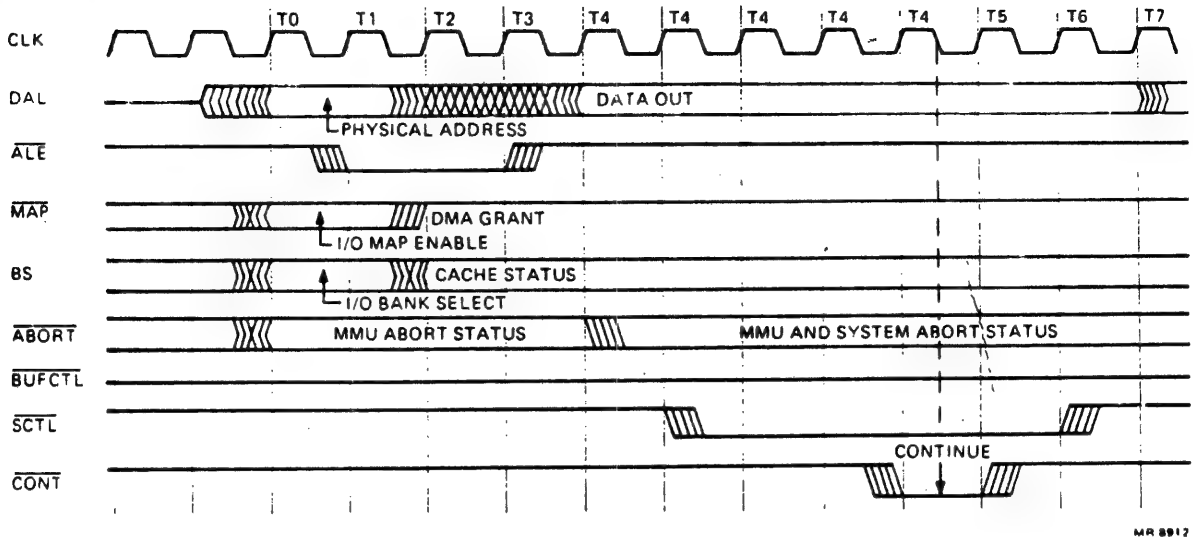


Figure 3-5 Bus Write Cycle

ALE typically latches the AIO code, the physical memory address on the DAL, the BS information, and the I/O map enable signal (MAP).

SCTL is asserted during the stretched portion of the cycle. The write data is valid when SCTL is asserted and the leading and trailing edges of SCTL can be used by external logic to latch this data. BUFCTL is not asserted during Bus Write cycles.

If an MMU error or address error abort occurs, the DCJ11 asserts ABORT during the first part of the cycle. Externally generated aborts must cause ABORT to be asserted during the stretched portion of the cycle.

NOTE

If an abort occurs during the first part of the cycle, the DAL, BS, and MAP information should be ignored for the remainder of the cycle.

During Bus Byte Write cycles, all 16 bits of DAL<15:0> are driven. If the address is even, the correct data is on the low byte. If the address is odd, the correct data is on the high byte. The data on the unused byte is unspecified.

Since a Bus Write cycle is always stretched, CONT must be asserted to end the cycle.

3.7 GENERAL-PURPOSE (GP) READ CYCLE

General-purpose read cycles allow the DCJ11 to read data from non-PDP-11 addressable external logic. A general-purpose read cycle involves the driving of an address on DAL<7:0> (called the general-purpose or GP code) which external logic must decode and respond to. General-purpose read cycles involve the reading of a full word. If the DCJ11 requires only a byte, it reads a word and ignores the unneeded byte. Timing for General-Purpose Read cycles is shown in Figure 3-6.

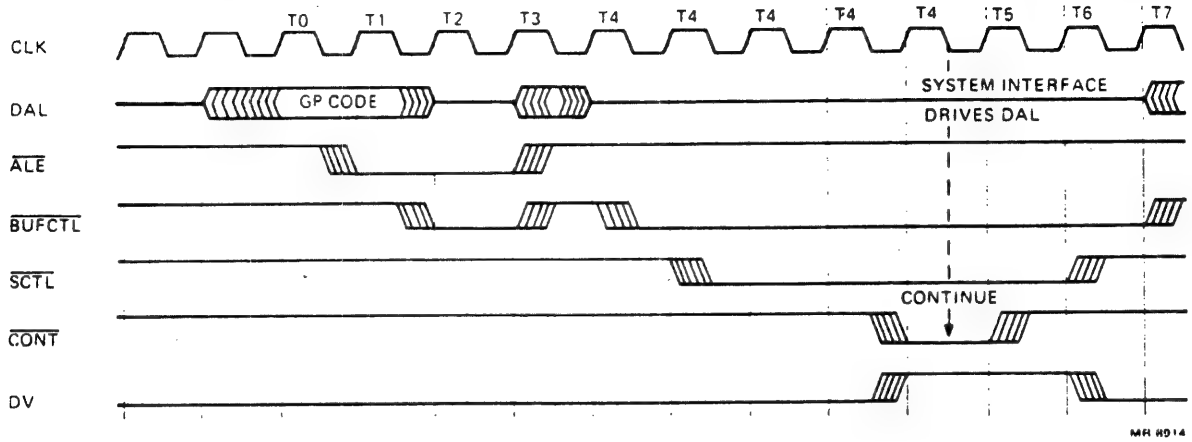


Figure 3-6 General-Purpose Read Cycle

$\overline{ALE}$ is typically used to latch the AIO code and the general-purpose code on the DAL. A GP Read is always stretched and thus always lasts a minimum of eight clock periods. The GP code (which specifies the source of the read data) is driven onto DAL<7:0> during the first part of the cycle. At this time, DAL<21:8> should be ignored. The general-purpose read codes are summarized in Table 3-2.

Table 3-2 General-Purpose Read Codes

Code	Function
000	Reads the power-up mode, HALT option, FPA option, POK, and boot address. See Chapter 8 - Interfacing for further details.
001	Reads FPA data (if FPA exists).
002	Reads the power-up mode, HALT option, FPA option, POK, and boot address, and clears FPA's FPS.
003	Acknowledges FPE and reads FEC (floating exception code) register (if FPA exists).

Note that GP Read data is latched into the DCJ11 both at the

rising edge of T3 and when DV is asserted during the stretched portion of the cycle (see Figure 3-6). Thus if the data is valid at the rising edge of T3, it is latched at that time and DV is not required. If the data is not valid at the rising edge of T3, DV is required to latch the valid data. Since a GP Read cycle is stretched, it must be ended by the assertion of CONT.

NOTE

General-Purpose Read cycles can not be aborted by the DCJ11 and should not be aborted by external logic.

3.8 GENERAL-PURPOSE (GP) WRITE CYCLE

General-Purpose Write cycles allow the DCJ11 to write data to non-PDP-11 external logic. A General-Purpose Write cycle involves the driving of an address on DAL<7:0> (called the general-purpose or GP code) which external logic must decode and respond to. GP write cycles involve the writing of either a word or a byte. Timing for General-Purpose Write cycles is shown in Figure 3-7.

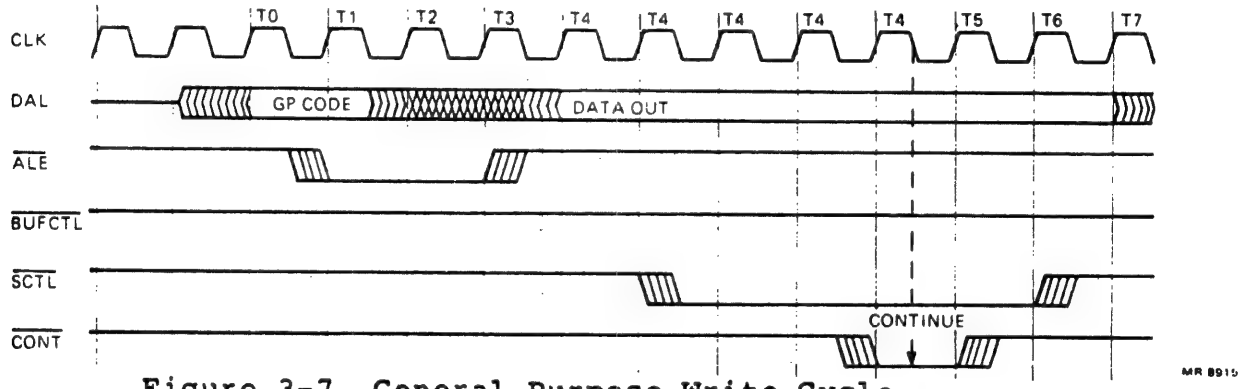


Figure 3-7 General-Purpose Write Cycle

ALE is typically used to latch the AIO code and the general-purpose code on the DAL. A GP Write is always stretched and thus always lasts a minimum of eight clock periods. The GP code (which specifies the destination of the write data) is driven onto DAL<7:0> during the first part of the cycle. At this time, DAL<21:8> should be ignored. Table 3-3 provides a summary of the GP Write codes. See Chapter 8 - Interfacing for further details.

Table 3-3 General-Purpose Write Codes

Code	Function
003	Writes FPA 16-bit data
014	Asserts bus reset signal
034	Indicates exit from console ODT
040	Reserved for future use
100	Acknowledges assertion of EVENT
140	Acknowledges Power Fail
214	Negates bus reset signal
220	Microdiagnostic test 1 passed
224	Microdiagnostic test 2 passed
230	Microdiagnostic test 3 passed
234	Indicates entry into console ODT

$\overline{SCTL}$ is asserted during the stretched portion of the GP Write cycle. The write data is valid (and can be latched) on the rising or falling edges of $\overline{SCTL}$. The write data is driven onto DAL<15:00>. Since a GP Write cycle is always stretched, it must be ended by the assertion of $\overline{CONT}$.

NOTE

General-Purpose Write cycles can not be aborted by the DCJ11 and should not be aborted by external logic.

3.9 INTERRUPT ACKNOWLEDGE BUS CYCLE

An Interrupt Acknowledge cycle (also called an Interrupt Vector Read cycle) is performed to service an interrupt request from IRQ<3:0>. Interrupt Acknowledge timing is illustrated in Figure 3-8. Note that the interrupt request on IRQ<3:0> must be deasserted by the end of the cycle.

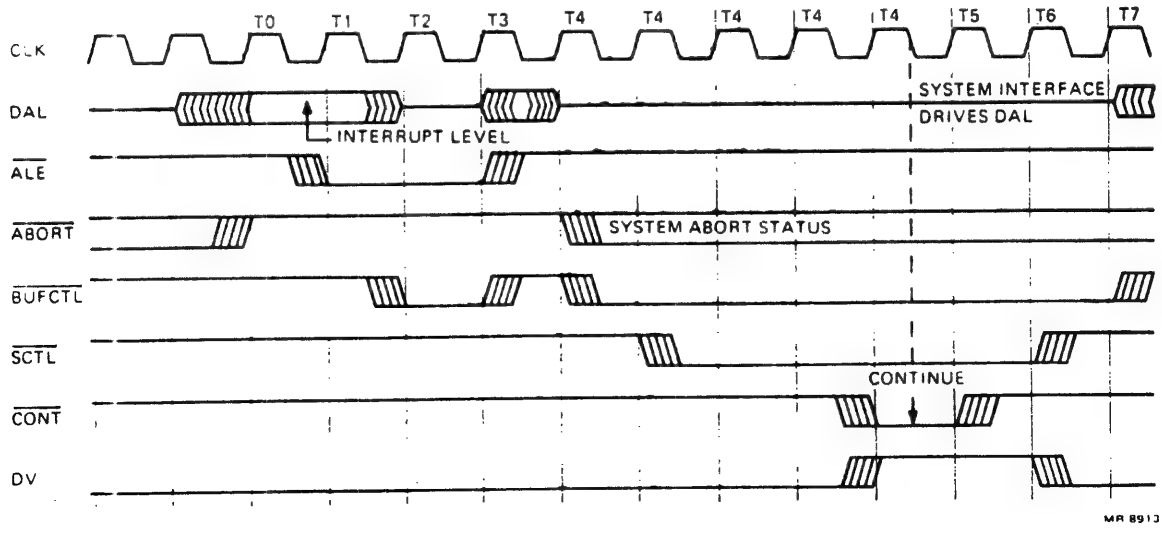


Figure 3-8 Interrupt Acknowledge Cycle

$\overline{ALE}$ is typically used by external logic to latch the AIO code and the acknowledged interrupt level. The interrupt level acknowledged is driven onto DAL<3:0> at the beginning of the cycle as shown in the table below.

Table 3-4 Interrupt Acknowledgement

DAL<3:0>	IRQ level acknowledged
0001	4
0010	5
0100	6
1000	7

At this time DAL<21:4>=0.

As shown in Figure 3-8, the interrupt vector address is placed on the DAL by the interrupting device during the second part of the cycle. An Interrupt Acknowledge cycle is always stretched and consists of at least eight clock periods. It is stretched in increments of two clock periods (T4) until the $\overline{CONT}$ input is asserted, at which time the cycle is ended.

Note that the interrupt vector is latched into the DCJ11 both at the rising edge of T3 and when DV is asserted during the stretched portion of the cycle. Thus if the interrupt vector is valid at the rising edge of T3, it is latched at that time and DV is not required. If the interrupt vector is not valid at the rising edge of T3, DV is required to latch it.

An Interrupt Acknowledge cycle can be aborted during the stretched portion of the cycle if $\overline{ABORT}$ is asserted by external logic. The DCJ11 does not assert $\overline{ABORT}$ during the first part of an Interrupt Acknowledge cycle. If an abort occurs, the DCJ11 ignores the interrupt request and continues execution.

3.10 DMA REQUESTS AND GRANTS

If external logic needs to use the DAL to transfer data, it must: (1) cause the DCJ11 to put the DAL in the high-impedance state, and (2) stretch the cycle currently in progress while external logic makes use of the DAL. This is accomplished by asserting the $\overline{DMR}$ input during the first part of a cycle. In response, the DMA request will be acknowledged and granted for all cycle types except Bus Write and GP Write cycles. During Write cycles (which are always stretched), the DAL carries write data during the second part of a cycle, during which time the DAL is not placed in the high-impedance state. External logic could be designed such that DMA transfers could occur during Write cycles as long as the DMA transfer did not use the DAL coming directly from the DCJ11 (a buffered version of the DAL could be used instead). In other words, external logic is not prevented from performing a DMA operation simply because a DMA grant does not occur.

A DMA request is acknowledged by asserting $\overline{MAP}$ during the second part of a cycle. A cycle involving a DMA transfer is stretched and thus lasts a minimum of eight clock periods. It will continue

to be stretched in increments of two clock periods until the $\overline{\text{CONT}}$ input is asserted. Note that the deassertion of $\overline{\text{DMR}}$ does not end the cycle.

4.1 INTRODUCTION

The DCJ11 contains a memory management unit (MMU) which provides the user with the hardware necessary to effect complete memory management and protection. The MMU is designed to provide access to all of physical memory and is an important part of multi-user, multiprogramming systems where memory protection and relocation facilities are necessary.

The MMU is used to assign segments of memory called pages to a user program and prevent that user from making unauthorized accesses to pages outside his assigned area. A user is thus prevented from accidental or willful destruction of any other user program or the system executive program.

The MMU is usually used in conjunction with a supervisory program which determines how the MMU is to operate. In multiprogramming environments this supervisory program controls the execution of the various user programs, manages the allocation of memory and peripheral device resources, and safeguards the integrity of the system as a whole by careful control of each user program.

The basic characteristics of the DCJ11 memory management unit are:

- 16 kernel mode memory pages
- 16 supervisor mode memory pages
- 16 user mode memory pages
- 8 pages in each mode for instructions
- 8 pages in each mode for data
- Page lengths from 64 to 8192 bytes
- Each page provided with full protection and relocation
- Transparent operation
- Memory access to 4 million bytes

The remainder of this chapter explains these characteristics in detail.

4.2 ADDRESSING

When the MMU is active, a 16-bit address referenced in a program is interpreted as a virtual address (VA) containing information to be used in constructing a new 22-bit physical address (PA). The information contained in the virtual address is combined with relocation information contained in a register called the page address register (PAR) to yield the 22-bit physical address. Using the MMU, memory can be dynamically allocated in pages composed of from 1 to 128 contiguous blocks of 64 bytes each . Figure 4-1 illustrates the relocation of virtual addresses to

physical addresses via page address registers.

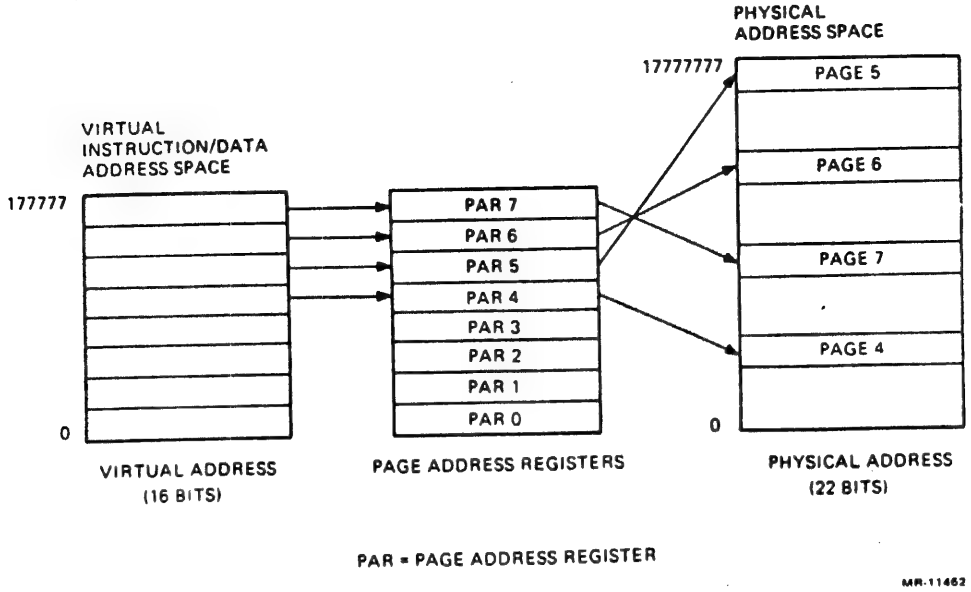


Figure 4-1 Virtual Address Mapping Into Physical Address

The starting physical address for each page is an integral multiple of 64 bytes, and each page has a maximum size of 8192 bytes. Pages may be located anywhere within the 22-bit physical address space.

Only one set of eight page address registers are illustrated in Figure 4-1. Actually, six such sets of page address registers are used by the MMU. The determination of which set of page registers is enabled at any given time depends on the current CPU mode of operation (i.e., kernel, supervisor, or user mode) and whether the MMU is mapping instructions (into I space) or data (into D space). Refer to Paragraph 4.5 for further details.

4.3 I SPACE AND D SPACE

When the MMU is active, all addresses are mapped into either instruction (I) space or data (D) space. I space is used for all instruction fetches, index words, absolute addresses and immediate operands. D space is used for all other references. I space and D space each have 8 PARs in each mode of CPU operation (kernel, supervisor, and user). Using memory management register #3 (MMR3), D space can be disabled such that all references (instruction and data) are mapped through I space.

Table 4-1 defines how memory references are mapped into the I and D spaces. Note that the determination of whether a memory reference gets mapped into I space or D space depends on: the type of instruction, the addressing mode, and the register selected.

Table 4-1 I and D Space Referencing
(first/second/third memory references)

Address Mode and Reg Select	Normal Instruction (not MTPI, MFPI MTPD, or MFPD)	MTPI, MFPI (PS<15:12> not 1111)	MTPD,MFPD, MFPI (PS<15:12> = 1111)
00 - 07	na	na	na
10 - 16	D	I	D
17	I	I	D
20 - 26	D	I	D
27	I	I	D
30 - 36	D/D	D/I	D/D
37	I/D	I/I	I/D
40 - 46	D	I	D
47	I	I	D
50 - 56	D/D	D/I	D/D
57	I/D	I/I	I/D
60 - 67	I/D	I/I	I/D
70 - 77	I/D/D	I/D/I	I/D/D

4.4 CONSTRUCTION OF A PHYSICAL ADDRESS

The basic information needed for the construction of a physical address comes from the virtual address (illustrated in Figure 4-2) and the appropriate PAR set.

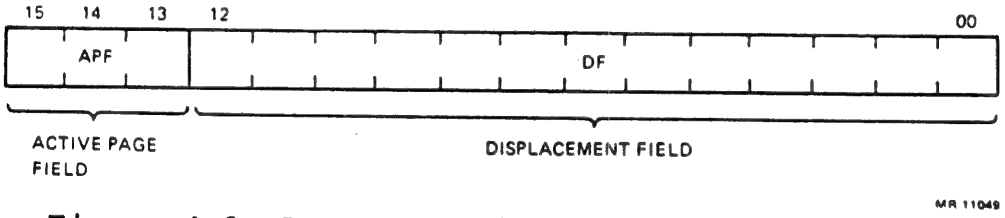


Figure 4-2 Interpretation of a Virtual Address

The virtual address consists of:

1. The active page field (APF). This 3-bit field determines which of eight page address registers (PAR0 through PAR7) will be used to form the physical address.

2. The displacement field (DF). This 13-bit field contains an address relative to the beginning of a page. This permits page lengths up to 8K bytes. The DF is further subdivided into two fields as shown in Figure 4-3.

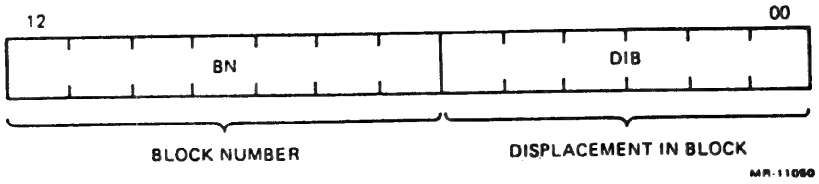


Figure 4-3 Displacement Field of Virtual Address

The displacement field (DF) consists of:

1. The block number (BN). This 7-bit field is interpreted as the block number within the current page.

2. The displacement in block (DIB). This 6-bit field contains the displacement of the address within the block specified by the block number.

The remainder of the information needed to construct the physical address comes from the 16-bit page address field (PAF) (i.e. the contents of the page address register (PAR)) that specifies the starting address of a particular memory page. The PAF is actually a block number in physical memory, e.g., PAF = 3 indicates a starting address of 192 (3 x 64 bytes per block) decimal or 300 octal in physical memory.

The formation of the physical address is illustrated in Figure 4-4.

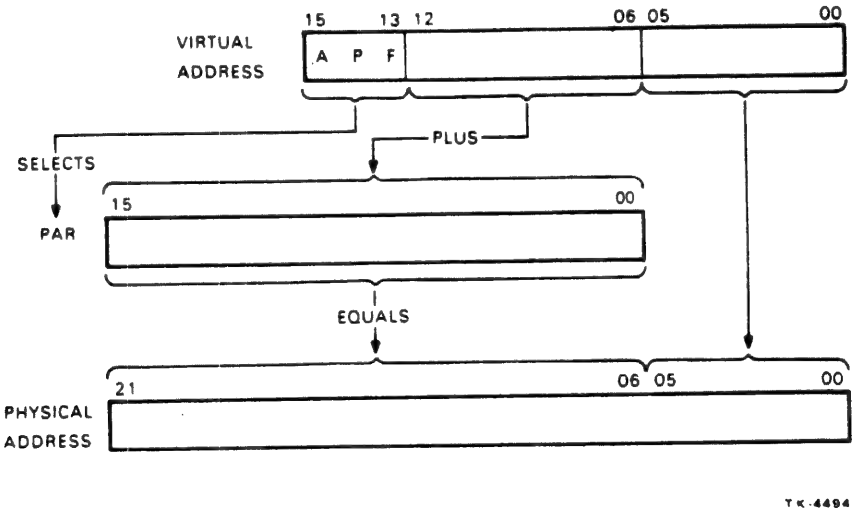


Figure 4-4 Construction of a Physical Address

The logical sequence involved in constructing a physical address is as follows:

1. Select a set of page address registers depending on the CPU mode (kernel, supervisor, or user) and the type of memory reference (I or D space).

2. Use the active page field (APF) from the virtual address to select one of eight page address registers (PAR0 through PAR7).

3. The page address field (PAF) of the selected page address register (PAR) contains the starting address of the currently active page as a block number in physical memory.

4. The block number (BN) from the virtual address is added to the page address field to yield the number of the block in physical memory which will contain the physical address being constructed.

5. The displacement in block (DIB) from the displacement field of the virtual address is appended to the physical block number to yield a true 22-bit DCJ11 physical address.

4.5 MANAGEMENT REGISTERS

The DCJ11 MMU implements three sets of 32 16-bit registers as shown in Figure 4-5. One set of registers is used in kernel mode, another in supervisor mode, and the other in user mode. The choice of which set to be used is determined by the current CPU mode contained in the processor status register (PS). Each set consists of two groups of 16 registers. One group is used for references to instruction (I) space and one to data (D) space. The I space group is used for all instruction fetches, index words, absolute addresses, and immediate operands. The D space group is used for all other references, providing D space has not been disabled by memory management register #3. Each group contains 8 pairs of 16-bit registers. Half of the registers in each group are page address registers, which operate as explained previously. The other registers are page descriptor registers (PDRs). PARs and PDRs are always selected in pairs. A PAR/PDR pair contains all the information needed to describe and locate a currently active memory page.

Each of the memory management registers described above are located in the uppermost 8K bytes of the physical address space (see Paragraph 4.9).

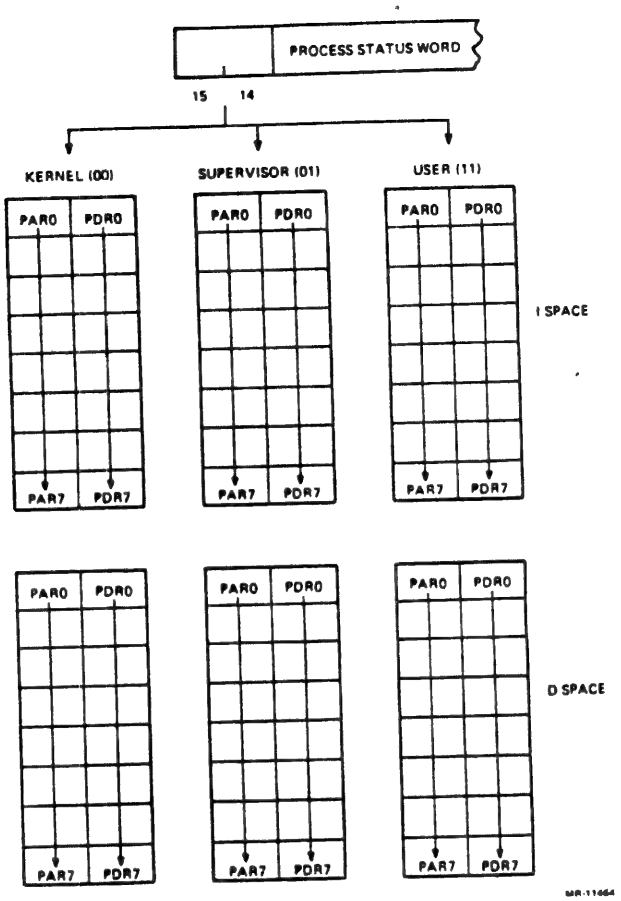


Figure 4-5 Active Page Registers

4.5.1 Page Address Registers (PARs) - As shown in Figure 4-6, each page address register contains a 16-bit page address field (PAF) which specifies the starting address of a page as a block number in physical memory.

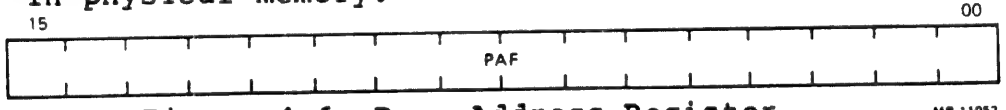


Figure 4-6 Page Address Register

The page address register which contains the page address field may be thought of as a relocation register containing a relocation constant, or as a base register containing a base address.

4.5.2 Page Descriptor Registers (PDRs) - Page descriptor registers (PDRs) contain information on page expansion direction, page length, and access control. Refer to Figure 4-7.

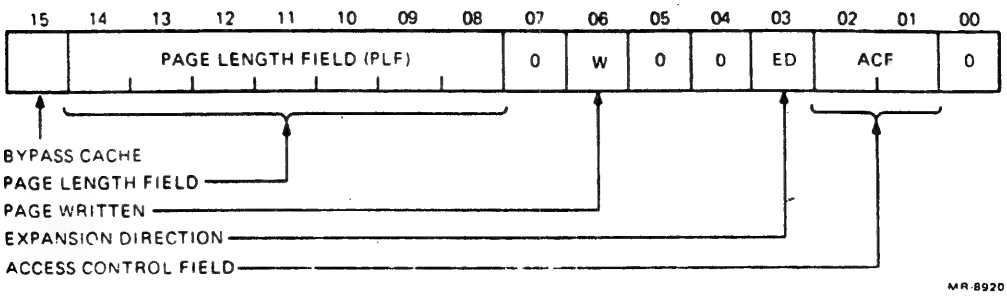


Figure 4-7 Page Descriptor Register (PDR)

4.5.2.1 Bypass Cache - Bit 15 implements a conditional cache bypass mechanism. If set, references to the selected virtual page can bypass cache memory if a cache is present in the system.

4.5.2.2 Page Length Field (PLF) - This 7-bit field occupying bits <14:8> of the PDR specifies the block number, which defines the boundary of that page. The block number of the virtual address is compared against the page length field to detect length errors. An error occurs when expanding upwards if the block number is greater than the page length field and when expanding downwards if the block number is less than the page length field.

4.5.2.3 Page Written - Bit 6 (the W bit) indicates whether or not this page has been modified (i.e., written into) since either the PAR or PDR was loaded (W = 1 means the page has been modified). The W bit is useful in applications which involve disk swapping and memory overlays. It is used to determine which pages have been modified and hence must be saved in their new form and which pages have not been modified and can simply be overlaid.

Note that the W bit is reset to 0 whenever either PAR or PDR is modified (written into).

4.5.2.4 Expansion Direction (ED) - Bit 3 specifies in which direction the page expands. If ED = 0 the page expands upwards from block number 0 to include blocks with higher addresses; if ED = 1 the page expands downwards from block number 127 to include blocks with lower addresses. Upward expansion is usually used for program space while downward expansion is usually used for stack space.

4.5.2.5 Access Control Field - This 2-bit field, occupying bits <2:1> of the page descriptor register contains the access rights of a particular page. The access codes or "keys" specify the manner in which a page may be accessed and whether or not a given access should result in an abort of the current operation. A memory reference which causes an abort must not be completed by the system interface. Aborts are used to catch "missing page faults", prevent illegal accesses, etc.

In the context of access control the term "write" is used to indicate the action of any instruction which modifies the contents of any addressable byte. "Write" is synonymous with what is sometimes called a "store" or "modify" in many computer systems.

The modes of access are as follows:

00	non-resident	abort all accesses
01	read-only	abort on write attempt
10	unused	abort all accesses
11	read/write	access

4.5.2.6 Reserved Bits - Bits 7, 5, 4, and 0 are spare and are always read as 0. These bits are reserved for possible future expansion.

4.6 INTERRUPT CONDITIONS UNDER MEMORY MANAGEMENT CONTROL

With the MMU enabled, all trap, abort, and interrupt vectors are considered to be in kernel mode virtual address space. When a trap, abort, or interrupt occurs, control is transferred according to a new program counter (PC) and processor status word (PS) contained in a two-word vector that is relocated through the kernel page address register set. The old PC and PS is pushed onto the R6 stack specified by bits <15:14> of the new PS (00 = kernel, 01 = supervisor, 11 = user). Bits <15:14> also determine the new PAR set. In this manner it is possible for a kernel mode program to have complete control over service assignments for all interrupt conditions since the interrupt vector is located in kernel space. The kernel program may assign the service of a trap, abort, or interrupt condition to a supervisor or user mode program by simply setting bits <15:14> of the new PS.

4.7 FAULT RECOVERY REGISTERS

Aborts generated by the MMU are vectored through kernel virtual location 250. Memory management registers #0, #1, #2, and #3 are used to determine why the abort occurred, and allow for easy program restarting. Note that an abort to a location which is itself an invalid address will cause another abort. Thus the kernel program must insure that kernel virtual address 250 is mapped to a valid address, otherwise a loop will occur which will

require console intervention.

4.7.1 Memory Management Register #0 (MMR0) - MMR0 contains error flags, the page number whose reference caused the abort, and various other status flags. The register is organized as shown in Figure 4-8.

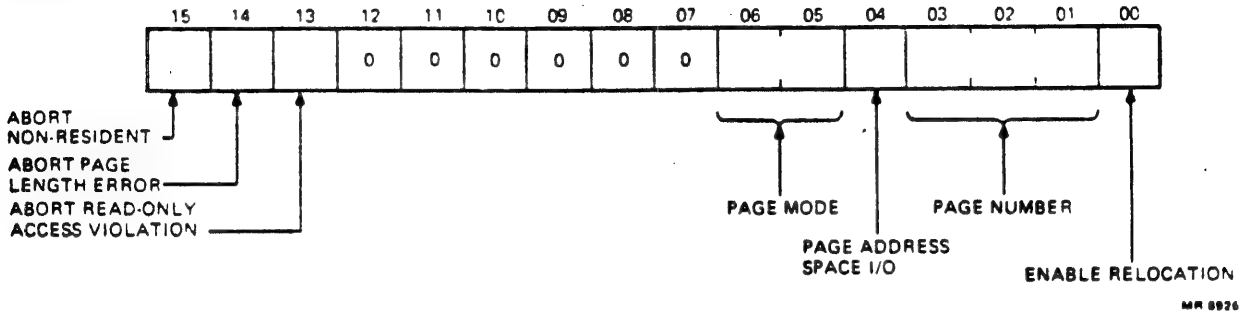


Figure 4-8 Memory Management Register #0 (MMR0)

4.7.1.1 Error Flags - Bits <15:13> are error flags. They may be considered to be in a "priority queue" in that flags to the right are less significant and should be ignored if a higher bit is set. That is, a non-resident fault service routine would ignore length and access control faults. A page length fault service routine would ignore access control faults.

Bits <15:13> when set (error conditions) cause the MMU to freeze the contents of MMR0 bits <6:1>, MMR1, and MMR2. This is to facilitate error recovery.

Bits <15:13> may be written under program control. No abort will occur, but the contents of the memory management registers will be frozen as in an abort.

Bits <15:13> are cleared at power-up, by a console start, or by a RESET instruction.

4.7.1.1.1 Abort -- Non-Resident - Bit 15 is set by attempting to access a page with an access control field key equal to 0 or 2. It is also set by attempting to use memory relocation with a processor mode of 2 (i.e., the illegal processor mode).

4.7.1.1.2 Abort -- Page Length - Bit 14 is set by attempting to access a location in a page with a block number (virtual address bits <12:6>) that is outside the area authorized by the page length field of the PDR for that page. Bits 14 and 15 may be set simultaneously by the same access attempt. Bit 14 may also be set by attempting to use memory relocation with a processor mode of 2.

4.7.1.1.3 Abort -- Read Only - Bit 13 is set by attempting to write in a "read-only" page. Read-only pages have access keys of 01.

4.7.1.2 Reserved Bits - Bits <12:7> are spare and are always read as 0. These bits are reserved for possible future expansion.

4.7.1.3 Processor Mode - Bits <6:5> indicate the CPU mode (kernel, supervisor, or user) associated with the page causing an abort (kernel = 00, supervisor = 01, user = 11, illegal mode = 10). If an illegal mode is specified, bit 15 is set.

4.7.1.4 Page Address Space - Bit 4 indicates the type of mapping (I or D) the MMU attempted when an abort occurred (0 = I space, 1 = D space). It is used in conjunction with bits <3:1>, page number.

4.7.1.5 Page Number - Bits <3:1> contain the page number of a reference causing an MMU abort. Note that pages, like blocks, are numbered from 0 upwards.

4.7.1.6 Enable Relocation - When bit 0 is set to a 1, the MMU is enabled and performs address relocation. When bit 0 is cleared, the MMU is inoperative and addresses are not relocated or protected. Bit 0 is cleared at power-up, by a console start, or by a RESET instruction.

4.7.2 Memory Management Register #1 (MMR1) - MMR1 (see Figure 4-9) records any autoincrement/autodecrement of the general-purpose registers, including references through the PC. This information is necessary to recover from an error resulting in an abort. MMR1 is cleared at the beginning of each instruction fetch. Whenever a general-purpose register is autoincremented or autodecremented, the register number and the amount (in 2's complement notation) by which the register was modified is written into MMR1. The low order byte of MMR1 is written first. It is not possible for a DCJ11 instruction to autoincrement/decrement more than two general-purpose registers per instruction before an "abort-causing" reference.

It is up to the software to determine which set of registers (kernel/supervisor/user -- general set 0/general set 1) was modified, by determining the CPU and register modes as contained in the PS at the time of the abort.

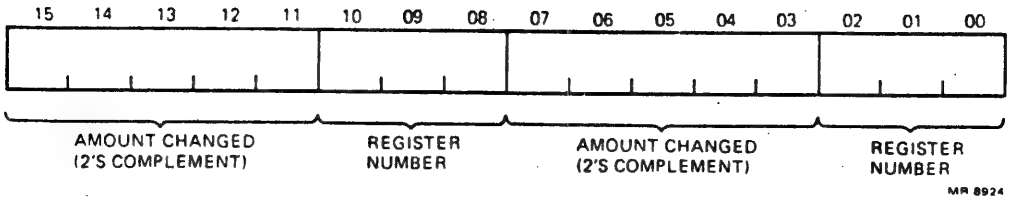


Figure 4-9 Memory Management Register #1 (MMR1)

4.7.3 Memory Management Register #2 (MMR2) - MMR2 is loaded with the current 16-bit virtual address at the beginning of each instruction fetch. MMR2 is read-only; it can not be written. MMR2 is the virtual program counter.

4.7.4 Memory Management Register #3 (MMR3) - As shown in Figure 4-10, MMR3 enables or disables the use of D space PARs and PDRs and 22-bit mapping and controls data on the time-multiplexed output MAP (pin 19 of the DCJ11).

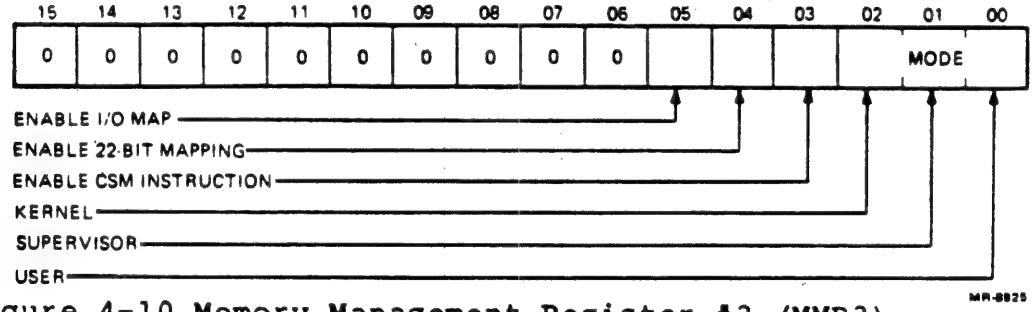


Figure 4-10 Memory Management Register #3 (MMR3)

4.7.4.1 Reserved Bits - Bits <15:6> are spare and are always read as 0. These bits are reserved for possible future expansion.

4.7.4.2 Enable I/O Map - Bit 5 is set to assert the MAP output of the DCJ11. If bit 5 = 1 MAP is asserted. If bit 5 = 0 MAP is unasserted. On initialization, MMR3 is cleared.

4.7.4.3 Enable 22-Bit Mapping - If bit 4 = 0 and the MMU is enabled (bit 0 of MMR0 = 1), the DCJ11 uses 18-bit mapping. If bit 4 = 1 and the MMU is enabled, the DCJ11 uses 22-bit mapping. If the MMU is disabled, bit 4 is ignored and 16-bit mapping is used. Figures 4-11, 4-12, and 4-13 illustrates the three mapping alternatives available.

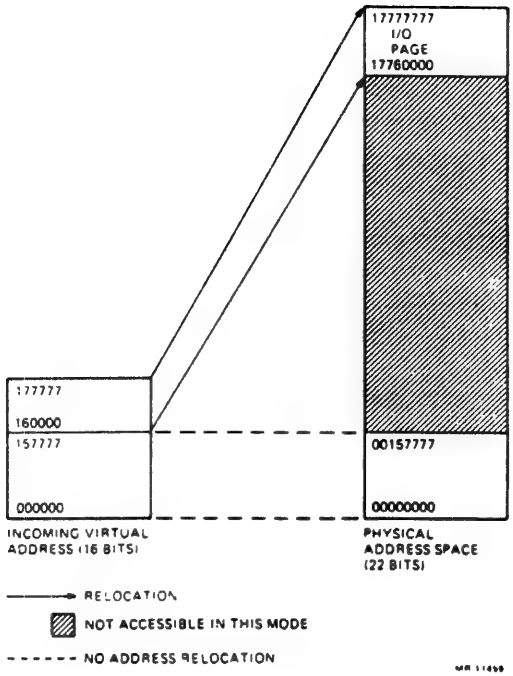


Figure 4-11 16-Bit Mapping

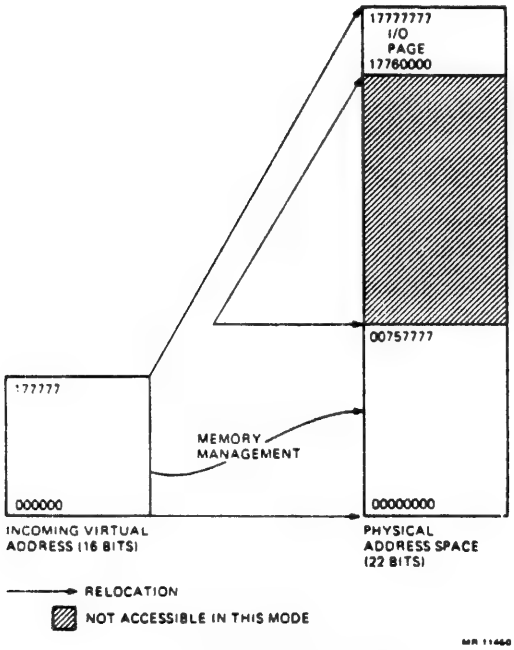


Figure 4-12 18-Bit Mapping

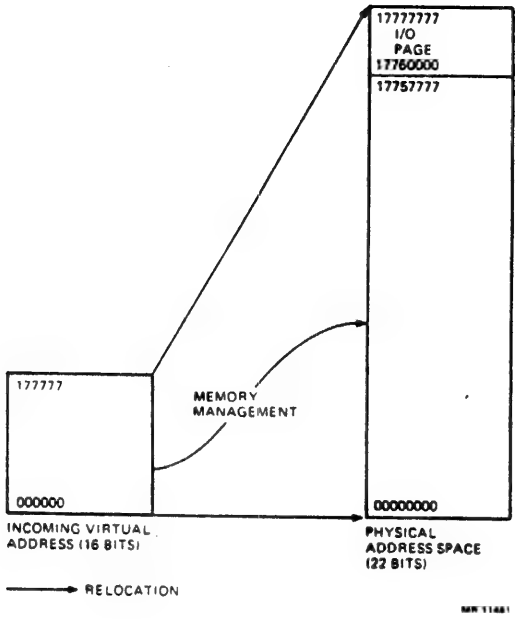


Figure 4-13 22-Bit Mapping

4.7.4.4 Enable Call To Supervisor Mode (CSM) Instruction - Bit 3 is used to enable a CSM instruction. If bit 3 is set to a 1, a CSM instruction will execute. If bit 3 = 0, a CSM instruction will cause a trap through vector location 10.

4.7.4.5 Kernel, Supervisor, And User Mode D Space Bits - Bits 2, 1, and 0 are the kernel, supervisor, and user mode D space bits, respectively. These bits determine whether D space mapping is enabled or disabled for each CPU mode. When D space is disabled, all memory references use the I space registers; when D space is enabled, both the I space and the D space registers are used. When a mode bit is set, D space is enabled; when a mode bit is clear, D space is disabled (see Table 4-2).

Table 4-2 Mode Bit Operations

BIT	STATE	OPERATION
2	0	Disable kernel D space
	1	Enable kernel D space
1	0	Disable supervisor D space
	1	Enable supervisor D space
0	0	Disable user D space
	1	Enable user D space

4.7.5 Instruction Back-Up/Restart Recovery - The process of "backing-up" and restarting a partially completed instruction involves:

1. Performing the appropriate memory management tasks to alleviate the cause of the abort (e.g., loading a missing page).

2. Restoring the general-purpose registers indicated in MMR1 to their original contents at the start of the instruction by subtracting the "modify value" specified in MMR1.

3. Restoring the PC to the "abort time" PC by loading R7 with the contents of MMR2, which contains the value of the virtual PC at the time the instruction generating the abort was fetched.

Note that this back-up/restart procedure assumes that the general-purpose register used in the aborted program segment will not be used by the abort recovery routine. This is automatically the case if the recovery program uses a different general register set.

4.7.6 Clearing Status Registers Following Abort - At the end of an abort service routine, bits <15:13> of MMR0 must be set to 0 to resume error checking. On the next memory reference following the clearing of these bits, the various memory management registers will resume monitoring the status of the addressing operations. MMR2 will be loaded with the next instruction address, MMR1 will store register change information, and MMR0 will log MMU status information.

4.7.7 Multiple Faults - Once an abort has occurred, any subsequent errors that occur will not affect the state of the memory management status registers. The information saved in MMR0, MMR1, MMR2, and MMR3 will always refer to the first abort that was detected.

4.8 MMU IMPLEMENTATION

The MMU is a very general purpose memory management tool. It can be used in a manner as simple or as intricate as desired. It can be anything from a simple memory expansion device to a very complete memory management facility.

In most normal applications, it is assumed that control over memory page assignments and their protection resides in a supervisory type program which operates at the nucleus of a CPU's executive (i.e. in kernel mode). It is further assumed that this kernel mode program would set access keys in such a way as to protect itself from willful or accidental destruction by supervisor mode or user mode programs. Facilities are also provided so that the nucleus can dynamically assign memory pages

of varying sizes in response to system needs.

4.8.1 Typical Memory Page - When the MMU is enabled, the kernel mode program, a supervisor mode program, and a user mode program each have eight active pages (described by the appropriate PARs and PDRs) for data, and eight for instructions. Each page is made up of from 1 to 128 blocks and is pointed to by the page address field of the corresponding PAR as illustrated in Figure 4-14.

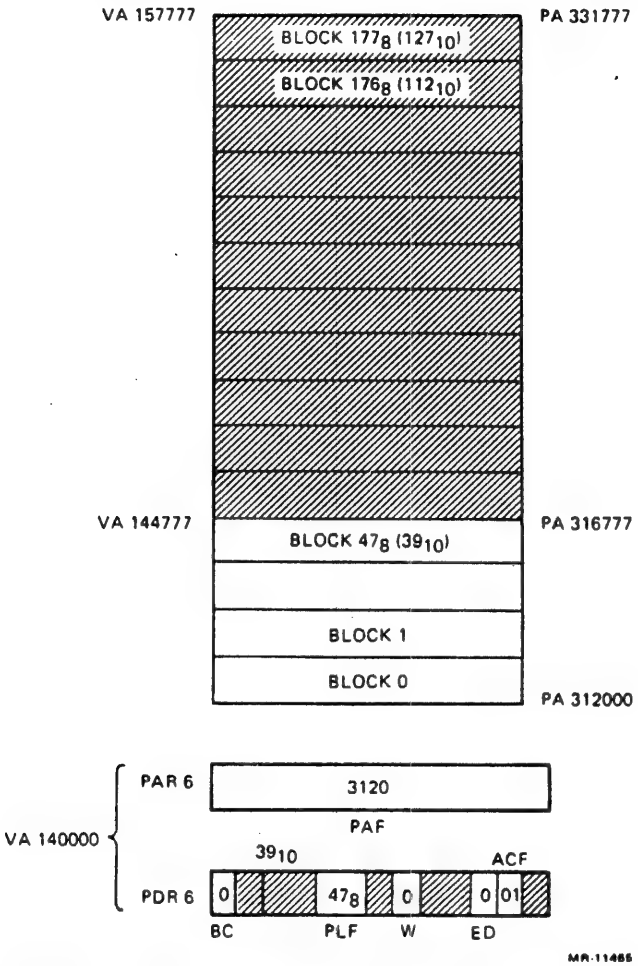


Figure 4-14 Typical Memory Page

The memory segment illustrated in Figure 4-14 has the following attributes:

1. Page length: 40 blocks.
2. Virtual address range: 140000 - 144777.
3. Physical address range: 312000 - 316777.
4. Nothing has been modified (i.e., written) in this page.
5. Read-only protection.

6. Upward expansion.

7. Cache (if present in the system) is not bypassed.

These attributes were determined according to the following scheme:

1. PAR6 and PDR6 were selected by the active page field of the virtual address. (Bits <15:13> of the virtual address = 110)

2. The initial address of the page was determined from the page address field of PAR6. (312000 (octal) = 3120 (octal) blocks x 64 (octal) bytes). Note that the PAR which contains the PAF constitutes what is often referred to as a base register containing a base address or a relocation register containing a relocation constant.

3. The page length (47 (octal) + 1 = 40 (decimal) blocks) was determined from the page length field contained in PDR6. Any attempts to reference beyond the 40 blocks in this page will cause a page length error which will result in an abort, vectored through kernel virtual address 250.

4. The physical addresses were constructed according to the scheme illustrated in Figure 4-4.

5. The W bit (W = 0) indicates that no locations in this page have been modified (i.e., written). If an attempt is made to modify any location in this particular page, an access control violation abort will occur. If this page were involved in a disk swapping or memory overlay scheme, the W bit would be used to determine whether it had been modified and thus required saving before overlay.

6. This page is read-only protected, i.e. no locations in this page may be modified. The mode of protection is specified by the access control field of PDR6.

7. The direction of expansion is upward (ED = 0). If more blocks are required in this segment, they will be added by assigning blocks with higher relative addresses.

8. The Bypass Cache bit (bit 15) = 0 which means that cache memory is not bypassed during this relocation operation.

Note that the various attributes which describe this page can all be determined under software control. The parameters describing the page are all loaded into the appropriate PAR and PDR under program control. In a normal application it is assumed that the particular page which itself contains these registers would be assigned to the control of a supervisory type program operating in kernel mode.

4.8.2 Non-Consecutive Memory Pages - It should be noted that although the correspondance between virtual addresses and PAR/PDR pairs is such that higher VAs have higher PAR/PDRs, this does not mean that higher virtual addresses necessarily correspond to higher physical addresses. It is quite simple to set up the PAFs of the PARs so that higher virtual address blocks may be located in lower physical address blocks as illustrated in Figure 4-15.

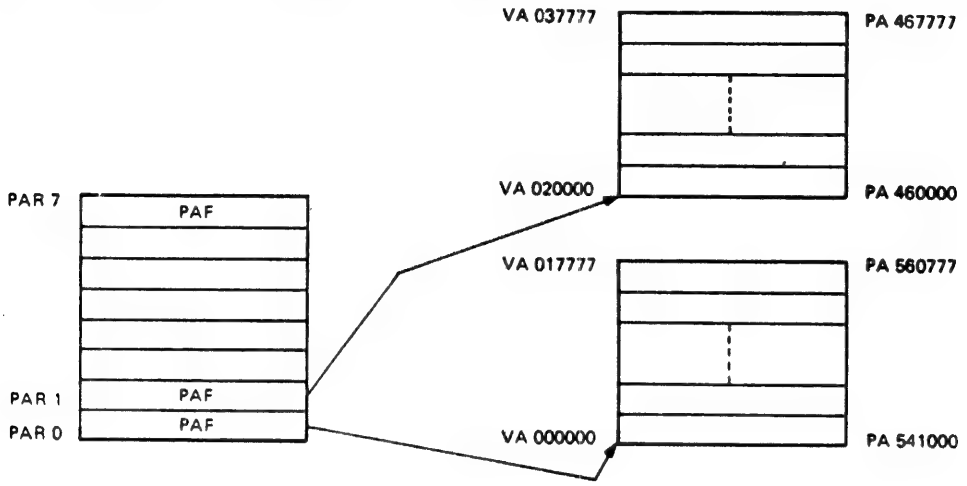


Figure 4-15 Non-Consecutive Memory Pages

Note that although a single memory page must consist of a block of contiguous locations, memory pages do not have to be located in consecutive physical address locations. Also note that the assignment of memory pages is not limited to consecutive non-overlapping physical address locations.

4.8.3 Stack Memory Pages - When constructing DCJ11 programs, it is often desirable to isolate all program variables from program instructions by placing them on a register-indexed stack. These variables can then be pushed or popped from the stack as needed. DCJ11 stacks expand linearly downward to lower addresses when data is pushed onto them. Thus, when a memory page which contains a stack needs more room, it must expand downward. Blocks with lower addresses relative to the current page must be added. This mode of operation is specified by setting the expansion direction (ED) bit of the appropriate PDR to a 1. Figure 4-16 illustrates a typical stack memory page.

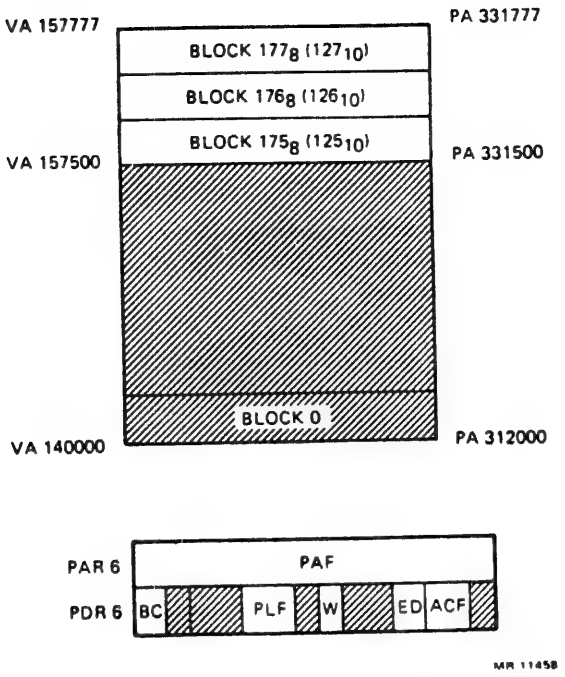


Figure 4-16 Typical Stack Memory Page

This page will have the following parameters:

- o PAR6: PAF = 3120
- o PDR6: PLF = 175 (octal) or 125 (decimal) (128 - 3).
- o ED = 1
- o W = 0 or 1
- o ACF = n (to be determined by the programmer as the need dictates)

Note: the W bit is set by internal chip hardware.

In this case the stack begins 128 blocks above the relative origin of this memory page and extends downward for a length of three blocks. A page length error abort vectored through kernel virtual address 250 will be generated by the MMU when an attempt is made to reference any location below the assigned area, i.e. when the block number from the virtual address is less than the page length field of the appropriate PDR.

4.8.4 Transparency - In a multiprogramming application memory pages can be allocated such that a particular program seems to have a complete 64K memory configuration. Using relocation, a kernel mode supervisory type program can easily perform all memory management tasks in a manner entirely transparent to a supervisor mode or user mode program. In effect, a DCJ11 system can be configured to provide maximum throughput and response to a variety of users each of which seems to have a powerful system all to himself.

4.9 MEMORY MANAGEMENT UNIT -- REGISTER MAP

REGISTER	ADDRESS
Memory Management Register #0 (MMR0)	17777572
Memory Management Register #1 (MMR1)	17777574
Memory Management Register #2 (MMR2)	17777576
Memory Management Register #3 (MMR3)	17772516
User I Space PDR0	17777600
.	.
.	.
.	.
User I Space PDR7	17777616
User D Space PDR0	17777620
.	.
.	.
.	.
User D Space PDR7	17777636
User I Space PAR0	17777640
.	.
.	.
.	.
User I Space PAR7	17777656
User D Space PAR0	17777660
.	.
.	.
.	.
User D Space PAR7	17777676
Supervisor I Space PDR0	17772200
.	.
.	.
.	.
Supervisor I Space PDR7	17772216
Supervisor D Space PDR0	17772220
.	.
.	.
.	.
Supervisor D Space PDR7	17772236
Supervisor I Space PAR0	17772240
.	.
.	.
.	.
Supervisor I Space PAR7	17772256

Supervisor D Space PAR0	17772260
.	.
.	.
.	.
Supervisor D Space PAR7	17772276
Kernel I Space PDR0	17772300
.	.
.	.
.	.
Kernel I Space PDR7	17772316
Kernel D Space PDR0	17772320
.	.
.	.
.	.
Kernel D Space PDR7	17772336
Kernel I Space PAR0	17772340
.	.
.	.
.	.
Kernel I Space PAR7	17772356
Kernel D Space PAR0	17772360
.	.
.	.
.	.
Kernel D Space PAR7	17772376

5.1 INTRODUCTION

This chapter discusses three special features incorporated into the DCJ11: cache memory status and control registers, console ODT, and pipeline processing hardware.

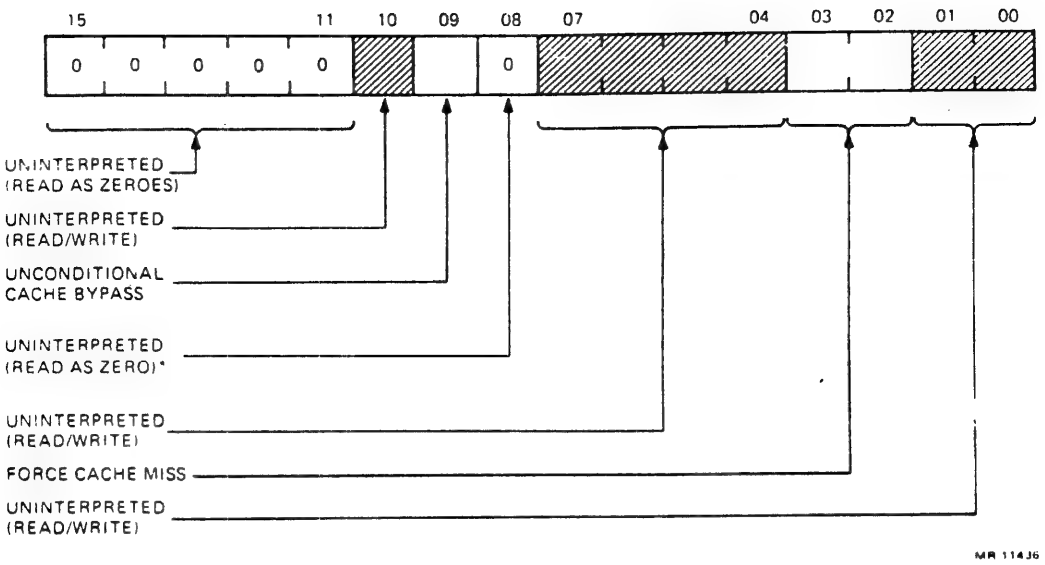
5.2 CACHE MEMORY STATUS AND CONTROL REGISTERS

The DCJ11 contains hardware that allows the user to incorporate cache memory into his system. This hardware consists of the cache control register and the hit/miss register. This hardware allows for a broad spectrum of cache implementations and the user has considerable flexibility in designing a cache memory scheme to fit his application. The paragraphs that follow not only describe the cache memory status and control registers in detail but also present some general considerations involved in designing cache memory into a DCJ11 based system. A sample cache memory implementation is also presented to illustrate a typical application of the cache memory status and control registers.

5.2.1 Cache Control Register - The cache control register (CCR) contains information which is used to control the operation of cache memory. It is accessed by referencing location 17777746. Only bits 9 and <3:2> of the CCR are interpreted by the DCJ11. Bits <10:0> are read/write bits. Bits <15:11,8> are always read as zeroes.

In order for the uninterpreted read/write bits (bits 10, <8:4>, and <1:0> to be used by external logic, the user must include a "shadow register" (write only) in his DCJ11 design. The shadow register simply retains a hardware accessible copy of the CCR information. Although the DCJ11 allows the reading and writing of CCR<10:0> and the writing of CCR<15:11>, changing bits <15:11>, 8, <7:4>, and <1:0> will have no hardware effect on the DCJ11.

CCR bits <15:11> are uninterpreted and always read as zeroes by the DCJ11 (see sample implementation in Paragraph 5.2.5). The user typically designs an external register for these bits if they must be interpreted. The format of the CCR is shown in Figure 5-1.



*Written as a logic 1 at power-up or when console ODT is started

Figure 5-1 Cache Control Register

5.2.1.1 Unconditional Cache Bypass (R/W) - When bit 9 is set to 1, all memory references access main memory, and all cache hits are invalidated.

5.2.1.2 Force Cache Miss (R/W) - When either of bits <3:2> is set to 1, all references are forced to main memory and all cache activity is suspended. This in effect disables the cache system.

5.2.1.3 Uninterpreted Bits - Bits <15:10>, <8:4>, and <1:0> are uninterpreted by the DCJ11. Bits 10, <8:4>, and <1:0> are read/write bits and bits <15:11> are always read as zeroes.

5.2.2 Hit/Miss Register - The Hit/Miss Register (HMR) indicates whether the six most recent CPU memory references resulted in cache hits or cache misses. It is accessed by referencing location 17777752. Refer to Figure 5-2. Bits <15:6> are always read as zeroes. Bits <5:0> are read-only bits. Bits enter from the right (at bit 0) and are shifted leftward. A logical one indicates a cache hit, and a zero indicates a cache miss. This register is used to help diagnose the cache system.

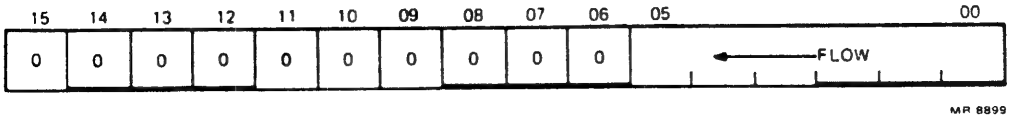


Figure 5-2 Hit/Miss Register

5.2.3 General Operation - Cache memory is typically a high-speed memory that buffers data between the CPU and main memory. When a memory access occurs, the system looks for data in the fast cache memory first. If found (a hit), the data is read or written to or from the cache and execution proceeds at the fastest rate. If not found (a miss), the data must be read from or written to main memory.

In a write-through cache system a CPU request to write data into memory causes data to be written to both the cache and to main memory. This is to insure that both stores are always updated immediately. PDP-11 systems with cache normally use the write-through technique.

Typical hit/miss operations in a write-through cache system are summarized in Table 5-1.

Table 5-1: Typical Hit/Miss Operations

	What Happens In	
	CACHE	MAIN MEMORY
READ		
hit	no change	no change
miss	updated	no change
WRITE		
hit	updated	updated
miss	no change	updated

In a typical program, WRITEs occur only 10-15% of the time and READs occur 85-90% of the time. Thus, READ misses cause the cache to be updated.

The I/O page of physical memory (the top 8K bytes) is not typically cached. This is because the I/O page contains device status registers which, when read, must always convey the latest information.

When a DMA device writes to a cached location, the overwritten cache entry is typically invalidated. The cache system monitors DMA transactions to determine if this action is needed.

There are several design parameters that must be considered when constructing a cache memory, cache size and block size to name but two. A detailed discussion of cache design is beyond the scope of this document, but an introduction to the subject is found in Section VI of the KB11-C Processor Manual (EK-KB11C-TM). An 8 KB direct mapped cache is presented as an implementation example in Paragraph 5.2.5.

5.2.4 Cache Memory In A Multiprocessor Environment - In a multiprocessor system where each processor has its own cache memory, care must be taken to avoid caching data that was invalidated by another processor ("stale" data). A simple software method can prevent this situation. Any shared address must bypass the cache, i.e., the reference must go to main memory, and if the address was previously cached, the entry must be invalidated. The DCJ11 provides three bypass mechanisms: an unconditional bypass in which every reference is bypassed; a conditional bypass in which bypassing is on a page-by-page basis; and finally, a selective bypass in which the bypassing is done during operand references. The unconditional bypass is selected by setting bit 9 of the Cache Control Register (see Paragraph 5.2.1). The conditional bypass is selected when bit 15 of the currently selected Page Descriptor Register PDR is set (see Paragraph 4.5.2). The selective bypass occurs during the operand references of the instructions used in multiprocessing functions (TSTSET, WRTLCK and ASRB).

5.2.5 Sample Implementation - The following is a description of the operation of an 8 Kb direct mapped cache with a block size of two bytes as implemented on a DCJ11 based system. This is only one of many possible implementations.

A direct mapped cache is organized such that each physical memory address is associated with a particular "block" of memory in the cache. In this case we have an 8 KB cache with a block size of two bytes. This means there are 4K blocks in the cache. Each word in physical memory is associated with one of these 4K blocks.

Consider each physical address as being made up of three parts (see Figure 5-3). The first part is bit zero. Bit zero specifies which of the two bytes in a two-byte block is to be accessed. The next part, bits <12:1>, is called the cache index and specifies which of the 4K blocks in the cache is to be accessed. The third part, bits <21:13>, is called the cache tag. One cache tag per block is stored in the cache to uniquely identify physical memory locations.

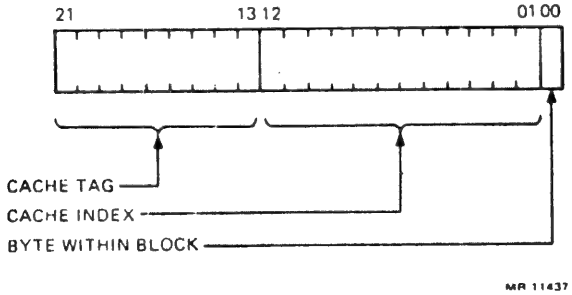


Figure 5-3 Physical Address Partitioning for Cache Memory

For example, if the DCJ11 accesses location 10002477, cache control logic (designed by the user) looks at the cache tag associated with the information currently in cache block number 1237 (bits <12:1>). If this cache tag is 400 (bits <21:13>), the cache control logic sends both bytes in that block to the DCJ11. Since bit 0 is a 1, the DCJ11 automatically selects the high byte

(the low byte is ignored). If the stored cache tag is not 400, the control logic fetches two bytes from memory (10002476 and 10002477), sends 10002477 to the DCJ11, loads the two bytes into cache block 1237, and changes the cache tag of that block to 400.

Any location whose cache index is 1237 will be loaded into block 1237 of cache memory. This is the only place the cache control logic has to look if the DCJ11 accesses the data from a location whose cache index is 1237.

Figure 5-4 illustrates a format for each cache block. The 9-bit cache tag is stored in bits <24:16> and the two bytes of data which comprise the block are stored in bits <15:0>. Bit 25 is a Valid Bit which indicates whether or not this cache block contains valid data. Data would be invalid for example immediately after power-up, and the cache control logic would clear the valid bit in this case.

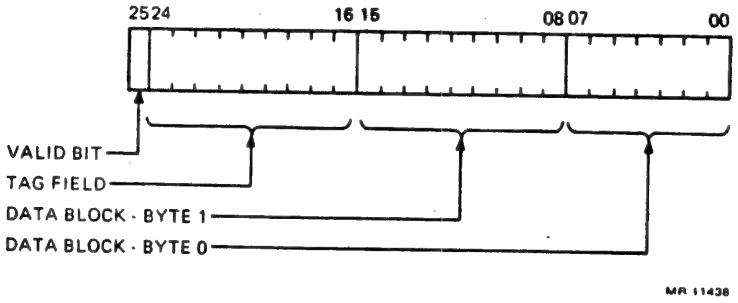


Figure 5-4 Cache Entry

Notice that only the cache tag of a location need be stored in a cache entry because only the cache tag is required to uniquely identify a location. The cache index need not be compared because anything stored in block 1237 (for example) is known to have bits <12:1> of its address set to 1237.

If desired, cache entries can also include parity information as shown in Figure 5-5.

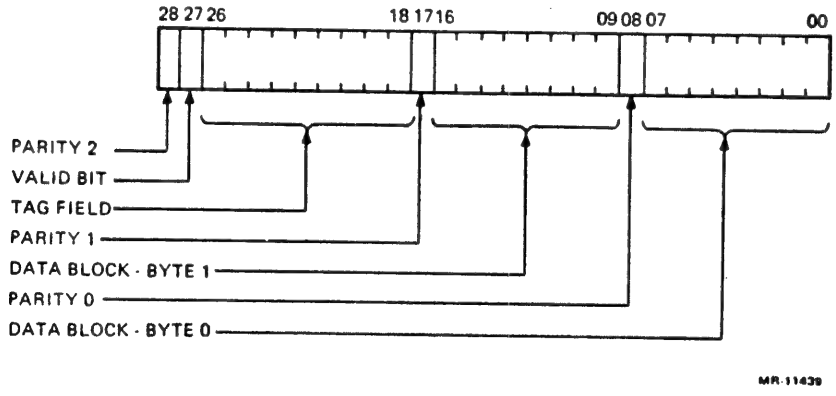


Figure 5-5 Cache Entry With Parity

The Parity 0 Bit stores parity information for byte 0, the Parity 1 Bit stores parity information for byte 1, and the Parity 2 Bit stores parity information for the cache tag/valid bit combination.

The Cache Control Register for this example is configured as shown

in Figure 5-6.

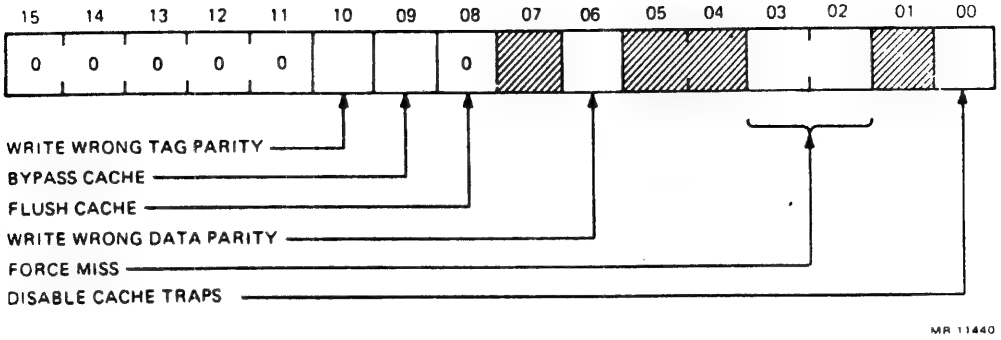


Figure 5-6 Sample Cache Control Register

BIT	NAME	FUNCTION
<15:11>	Not Used (read as zeroes)	These bits are not used in this example. The DCJ11 will ignore any data written to these bits and will always read these bits as zeroes.
10	Write Wrong Tag Parity (read/write)	This bit, when set, causes the cache tag parity bit (Parity 2) to be written with wrong parity when a cache entry is updated (i.e. upon CPU read misses and write hits). This causes a cache tag parity error on the next access to a location referenced by the entry.
9	Bypass Cache (read/write)	This bit, when set, forces all CPU memory references to go directly to main memory. Read or write hits will result in invalidation of accessed locations in the cache.
8	Flush Cache (read as zero)	Setting this bit causes the entire contents of the cache to be declared invalid. Writing a "0" into this bit will have no effect.
7	Not Used (read/write)	This bit is not used in this example.
6	Write Wrong Data Parity (read/write)	This bit, when set, causes the parity bits of the two data bytes (Parity 0 and Parity 1) to be written with wrong parity when

		updated (i.e. upon CPU read misses and write hits). This causes a cache parity error to occur on the next access to a location referenced by the entry.
<5:4>	Not Used (read/write)	These bits are not used in this example.
<3:2>	Force Miss (read/write)	These bits, when either is set, force all DCJ11 memory references to go directly to main memory. Unlike cache bypasses, force misses have no effect on cache entries. Enabling force miss effectively removes cache memory from the system.
1	Not Used (read/write)	These bits are not used in this example.
0	Disable Cache Traps (read/write)	This bit, when set, disables cache parity interrupts. When this bit is cleared, an interrupt occurs when a parity error is encountered.

All words read from the cache are checked for parity. A parity error in the accessed word causes the following CPU responses:

CCR<0>	Action
0	Interrupt through vector 114 and force miss.
1	Force miss only.

The CCR is cleared on power-up or by a console start. It is unaffected by a RESET instruction.

The cache response matrix for this example would be:

	CPU		DMA	
	Hit	Miss	Hit	Miss
Read	Read cached data	Read memory & allocate cache	Read memory	Read memory
Write	Write thru cache to memory	Write memory	Invalidate cache & write mem	Write memory
Read bypass	Invalidate cache & read mem	Read memory	na	na
Write bypass	Invalidate cache & write mem	Write memory	na	na
Read forced miss	Read memory	Read memory	na	na
Write forced miss	Write memory	Write memory	na	na

na = not applicable

5.3 CONSOLE ODT

The console octal debugging technique or console ODT allows the DCJ11 to respond to commands and information entered via a user-designed console terminal interface. The interface bus uses addresses 17777560 through 17777566 to communicate with console ODT. These addresses are generated in the DCJ11 and cannot be changed. Console ODT is a very useful aid in running and debugging programs. Communication between the user and DCJ11 is via a stream of ASCII characters which are interpreted by the DCJ11 as console commands. These commands are a subset of the commands used in DIGITAL's ODT-11 software for minicomputers.

5.3.1 Terminal Interface - The minimum optional hardware requirements for an interface permitting communication with console ODT are outlined in the paragraphs that follow (these requirements are met by the DLART DL-compatible asynchronous receiver/transceiver peripheral chip - DIGITAL Part No. DC319-AA).

5.3.1.1 Receiver Control/Status Register (RCSR) - The RCSR (Figure 5-7) must exist at address 17777560 for character input to console ODT. Console ODT does not execute output bus cycles to this address; therefore the RCSR only needs to respond to input bus cycles. System software may affect certain bits, such as Interrupt Enable (bit 6), but console ODT ignores this.

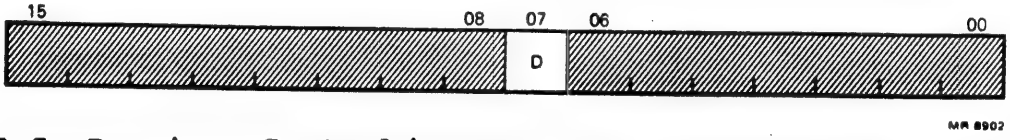


Figure 5-7 Receiver Control/Status Register (RCSR) - Address 17777560

Bit	Description
<15:8>	Unused. These bits may be in any state since console ODT does not use them.
<7>	Done flag. After a character is received and exists in the receiver buffer register (RBUF), the Done flag must be set to a 1. When the character is read from RBUF Done flag must be cleared by hardware.
<6:0>	Unused. These bits may be in any state since console ODT does not use them.

5.3.1.2 Receiver Buffer Register (RBUF) - The RBUF (Figure 5-8) must exist at address 17777562 for character input to console ODT. This register only needs to respond to input bus cycles since console ODT does not execute output bus cycles to this address. System software operates similarly, but DIGITAL diagnostics may cause output cycles and thus may not operate properly.

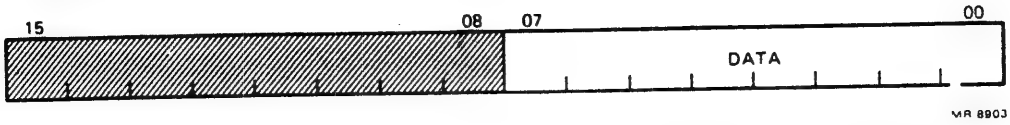


Figure 5-8 Receiver Buffer Register (RBUF) - Address 17777562

Bit	Description
<15:8>	Unused. These bits can be in any state since console ODT does not use them.
<7:0>	ASCII character. These eight bits are read by the processor and interpreted as a console ODT command. When bit 7 of RCSR is a 1, the processor reads data from the RBUF. After the input cycle, the hardware must clear bit 7 of RCSR to 0.

5.3.1.3 Transmitter Control And Status Register (XCSR) - The XCSR (Figure 5-9) must exist at address 17777564 for character output from console ODT. ODT does not execute output bus cycles to this address; therefore, the XCSR only needs to respond to input bus cycles. System software may cause output cycles to affect certain bits, such as Interrupt Enable, but console ODT ignores this.

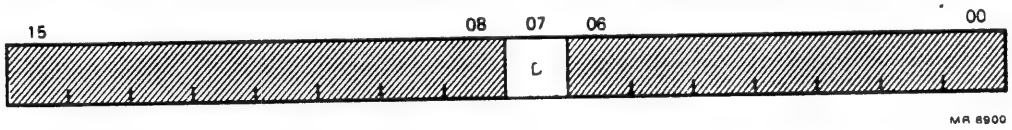


Figure 5-9 Transmitter Control/Status Register(XCSR) - Address 17777564

Bit	Description
<15:8>	Unused. These bits may be in any state since console ODT does not use them.
<7>	Done flag. In the idle state, this bit is a 1 indicating that the XBUF is ready to receive a character. After an output cycle to the transmitter buffer register (XBUF) by the processor, this bit must be cleared to 0 by the hardware. When the XBUF is ready to receive another character, the hardware sets this bit to 1.
<6:0>	Unused. These bits may be in any state since console ODT does not use them. Note that these bits may be meaningful to other DIGITAL interfaces.

5.3.1.4 Transmitter Buffer Register (XBUF) - The XBUF (Figure 5-10) must exist at address 17777566 for character output from console ODT. This register only needs to respond to output bus cycles since console ODT does not execute input bus cycles to this address. System software operates similarly but DIGITAL diagnostics may cause an input cycle and thus may not operate properly.

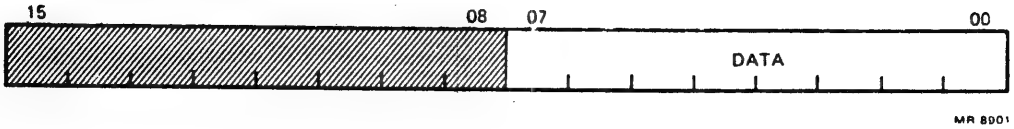


Figure 5-10 Transmitter Buffer Register (XBUF) - Address 17777566

Bit	Description
<15:8>	Unused. These bits may be in any state since console ODT does not use them.
<7:0>	ASCII character. These eight bits are written by the processor with the ASCII character output by ODT. When bit 7 of XCSR is a 1, the processor may perform an output cycle to XBUF.

5.3.2 Console ODT Operation - Console ODT operates the console terminal interface in half-duplex mode. Communication between console ODT and the interface is accomplished via programmed I/O techniques rather than interrupts. When console ODT is outputting characters using the transmit side of the interface, the receive side of the interface is not monitored for incoming characters. Any characters coming in at this time are lost. Console ODT does not check for error bits in the interface. If another processor is at the other end of the interface, that processor must operate within the format of half-duplex transmission. No input characters should be sent until console ODT has finished outputting.

5.3.2.1 Console ODT Initialization - Console ODT operation is initiated by any of the following:

1. Execution of a HALT instruction in kernel mode (if kernel HALT is enabled).
2. Assertion of the HALT signal on the system bus. The signal must be asserted long enough so that it is seen by the processor at the end of the current macroinstruction
3. At power-up, if the appropriate power-up option is selected.

Console ODT Input Sequence

The Console ODT entry sequence is as follows:

1. Output <CR><LF> to XBUF.
2. Output the contents of PC in six digits to XBUF.
3. Read and ignore character in RBUF. (May be a program character.)
4. Output <CR><LF> to XBUF.
5. Output the prompt character, @, to XBUF.
6. Enter a wait loop for input. The Done flag, bit 7 in RCSR, is tested. If it is 0, the test continues.
7. If RCSR bit 7 is a 1, then the low byte of RBUF is read.

5.3.2.2 Console ODT Output Sequence -

Console ODT does the following when it has a character ready for output:

1. Test XCSR bit 7 (Done flag) and if a 0, continue testing.

2. If XCSR bit 7 is a 1, write character to low byte of XBUF (high byte should be ignored by interface).

5.3.3 Console ODT Command Set - The console ODT command set is a subset of ODT-11 and uses the same command characters. Only specific characters are recognized as valid inputs; other inputs invoke a "?" response. The commands are summarized in Table 5-2.

The word "location," as used in the paragraphs that follow refers to a memory location, an I/O device register, an internal processor register, or the processor status word (PS).

Table 5-2 Console ODT Commands

Command	Symbol	Function
Slash	n/	Opens the specified location (n) and outputs its contents. n is an octal number.
Carriage Return	<CR>	Closes an open location.
Line Feed	<LF>	Closes an open location and then opens the next contiguous location.
Internal Register Designator	$n or Rn	Opens a specific processor register (n). n is an integer from 0 to 7 or the character S.
Processor Status Word Designator	S	Opens the PS - must follow an $ or R command.
Go	G	Starts program execution.
Proceed	P	Resumes execution of a program.
Binary Dump	Control-Shift-S	Manufacturing use only.

The parity bit (bit 7) on all input characters is ignored (i.e., not stripped) by console ODT. If an input character is echoed, the state of the parity bit is copied to the output buffer (XBUF). Output characters internally generated (e.g., <CR>) by ODT have the parity bit equal to 0. All commands are echoed except for ASCII codes in the range 0-17 (octal). Where applicable, the upper- and lowercases of command characters are recognized.

NOTE

In the examples that follow, the response from the processor is underlined, while the user's entry is not. When the user inputs an address or data, leading zeroes are not required. The DCJ11, however, outputs 8 digit octal addresses and 6 digit octal data words.

5.3.3.1 / (ASCII 057) Slash - This command is used to open a memory location, I/O device register, internal processor register, or processor status word and must be preceded by other characters which specify a location. In response to /, console ODT prints the contents of the location (i.e., six characters) and then a space (ASCII 40). After printing is complete, console ODT waits for either new data for that location or a valid close command.

Example: @001000/012525<SPACE>

where:

@	=	console ODT prompt character.
001000	=	octal location desired by the user (leading 0s are not required).
/	=	command to open and print contents of location.
012525	=	contents of octal location 1000.
<SPACE>	=	space character generated by console ODT.

5.3.3.2 <CR> (ASCII 015) Carriage Return - This command is used to close an open location. If a location's contents are to be changed, the user should precede the <CR> with the new data. If no change is desired, <CR> closes the location without altering its contents.

Example: @R1/004321<SPACE> <CR> <CR><LF>
@

Processor register R1 was opened and no change was desired so the user issued<CR>. In response to the <CR>, console ODT printed <CR><LF>@.

Example: @R1/004321<SPACE> 1234 <CR> <CR><LF>
@

In this case the user desired to change R1, so new data, 1234, was entered before issuing the <CR>. Console ODT deposited the new data in the open location and then printed <CR><LF>@.

Console ODT does not directly echo the <CR> entered by the user but instead prints a <CR>, followed by an <LF>, and @.

5.3.3.3 <LF> (ASCII 012) Line Feed - This command is used to close an open location and then open the next contiguous location. Memory locations and processor registers are incremented by 2 and 1 respectively. If the PS is open when a <LF> is issued, it is closed and a <CR><LF>@ is printed; no new location is opened. If the open location's contents are to be changed, the new data should precede the <LF>. If no data is entered, the location is closed without being altered.

Example: @R2/123456<SPACE> <LF> <CR><LF>
R3/054321<SPACE>

In this case, the user entered <LF> with no data preceding it. In

response, console ODT closed R2 and then opened R3. When a user has the last register, R7, open, and issues <LF>, console ODT opens the beginning register, R0.

Example:
```
@R7/000000<SPACE> <LF> <CR><LF>
R0/123456<SPACE>
```

Unlike with most other commands, console ODT does not echo the <LF>. Instead it prints <CR>, then <LF>, so that terminal printers operate properly. In order to make this easier to decode, console ODT does not echo ASCII characters in the range 0 - 17 (octal).

5.3.3.4 $ (ASCII 044) Or R (ASCII 122) Internal Register Designator - Either character when followed by a register number, 0 to 7, or PS designator, S, will open that specific processor register.

The $ character is recognized to be compatible with ODT-11. The R character was introduced because it can be conveniently typed with one key stroke and because it is an easily remembered symbol for a register.

Example:
```
@$0/000123<SPACE>
```

or

```
@R7/000123<SPACE> <LF>
R0/054321<SPACE>
```

If more than one character is typed after the R or $, console ODT uses the last character typed as the register designator.

5.3.3.5 S (ASCII 123) Processor Status Word - This designator is for opening the PS (processor status word) and may be employed only after the user has entered an R or $ register designator.

Example:
```
@RS/100377<SPACE> 0 <CR> <CR><LF>
```

NOTE

The trace bit (bit <4>) of the PS cannot be modified by the user. This is done so that PDP-11 program debugging utilities (e.g., ODT-11), which use the T bit for single-stepping, are not accidentally harmed by the user.

If the user issues a <LF> while the PS is open, the PS is closed and ODT prints <CR><LF>@. No new location is opened in this case.

5.3.3.6 G (ASCII 107) Go - This command is used to start program execution at a location entered immediately before the G. This function is equivalent to the LOAD ADDRESS and START switch sequence on other PDP-11 consoles.

Example: @200G<NULL><NULL>

The console ODT sequence for a G, after echoing the command character, is as follows.

1. Print two nulls (ASCII 0). This is intended to prevent the G character from getting flushed during the bus initialization sequence that follows, assuming a double-buffered UART chip is used in the console terminal interface.

2. Load R7 (PC) with the entered data. If no data is entered, 0 is used. (In the above example, R7 is set to 200, and that is where program execution begins.)

3. The PS, MMR0<15:13,0>, MMR3, PIRQ, CPU Error Register, Memory System Error Register, Cache Control Register, and Floating Point Status Register are cleared to zero.

4. The cache, if present, is flushed (if so implemented).

5. The system bus is initialized by the processor.

6. The service state is entered by the DCJ11. Any outstanding service requests are processed. If the bus HALT signal is asserted, the processor reenters the console ODT state. This feature is used to initialize a system without starting a program (R7 is altered).

5.3.3.7 P (ASCII 120) Proceed - This command is used to resume execution of a program and corresponds to the CONTINUE switch on other PDP-11 consoles. No programmer-visible machine state is altered using this command.

Example: @P

Program execution resumes at the address pointed to by R7. After the P is echoed, the DCJ11 immediately fetches the next instruction. After the instruction is executed, outstanding interrupts, if any, are serviced. If the HALT bus signal is asserted, it is recognized at the end of the instruction, and the DCJ11 enters the console ODT state. Upon entry, the content of the PC (R7) is printed. In this fashion, the user can single-instruction step through a program and obtain a PC "trace" on the terminal.

5.3.3.8 Control-Shift-S (ASCII 023) Binary Dump - This command is used for manufacturing test purposes and is not a normal user command. It is described here to explain the processor's response if accidentally invoked. It is intended to more efficiently display a portion of memory compared to using the "/" and <LF> commands. The protocol is as follows.

1. After a prompt character, console ODT receives a control-shift-S command and echoes it.

2. The host system at the other end of the serial line must send two 8 bit bytes which console ODT interprets as a starting address. These two bytes are not echoed.

   The first byte specifies starting address <15:08> and the second byte specifies starting address <07:00>. Address bits <21:16> are always forced to be 0; the dump command is restricted to the first 32K words of address space.

3. After the second address byte has been received, console ODT outputs ten bytes to the serial line starting at the address previously specified. When the output is finished, console ODT prints <CR><LF>@.

   If a user accidentally enters this command, it is recommended in order to exit from the command that two @ characters (ASCII 100) be entered as a starting address. After the binary dump, an @ prompt character is printed.

5.3.4 Address Specification - All I/O addresses (17760000 to 17777777) must be entered by the user with all 22 bits specified. For example, if a user desires to open the RCSR of the console serial interface he must enter 17777560, not 177560, or 777560.

5.3.4.1 General Registers - Whenever R0-R5 are referenced in console ODT, they access the general register set currently specified by PS bit 11 (PS<11>). If a program operating in general register set zero (PS<11> = 0) is halted and a general register is opened, register set zero is accessed. Similarily, if a program is operating in register set one, references to R0-R5 access register set one.

If a specific register set is desired, PS<11> must be set by the user to the appropriate value, and then the R0 through R5 commands can be used. If an operating program has been halted, the original value of PS<11> must be restored in order to continue execution.

Example: PS = 000000

@R4/052525<SPACE> <CR> <CR><LF>

R4 in register set zero has been opened.

```
@RS/000000<SPACE> 4000 <CR> <CR><LF>
@R4/177777<SPACE> <CR> <CR><LF>
@RS/004000<SPACE> 0 <CR> <CR><LF>
@P
```

In this case, R4 in register set one was desired. The PS was opened, and PS<11> was set to 1 (register set one). Then R4 was examined and closed. The original value of PS<11> was restored, and the program was continued using the P command.

5.3.4.2 Stack Pointers - Whenever R6 is referenced in console ODT, it accesses the stack pointer specified by the PS current mode bits (PS<15:14>). If a program operating in kernel mode (PS<15:14> = 00) is halted and R6 is opened, the kernel stack pointer is accessed. Similarly, if a program is operating in supervisor or user mode, R6 accesses the supervisor or user stack pointers.

If a specific stack pointer is desired, PS<15:14> must be set by the user to the appropriate value and then the R6 command can be used. If an operating program has been halted, the original value of PS<15:14> must be restored in order to continue execution.

Example: PS = 140000

```
@R6/123456<SPACE> <CR> <CR><LF>
```

The user mode stack pointer has been opened.

```
@RS/140000<SPACE> 0 <CR> <CR><LF>
@R6/123456<SPACE> <CR> <CR><LF>
@RS/000000<SPACE> 140000<CR> <CR><LF>
@P
```

In this case, the kernel mode stack pointer was desired. The PS was opened, and PS<15:14> were set to 00 (kernel mode). Then R6 was examined and closed. The original value of PS<15:14> was restored, and then the program was continued using the P command.

5.3.4.3 Floating Point Accumulators - The floating point accumulators cannot be accessed from console ODT. Only floating point instructions can access these registers.

5.3.5 Entering Octal Digits - When the user is specifying an address, console ODT will use the last eight octal digits if more than eight have been entered. When the user is specifying data, console ODT will use the last six octal digits if more than six have been entered. The user need not enter leading 0s for either address or data; console ODT forces 0s as the default. If an odd address is entered, console ODT responds to the error by printing ?<CR><LF>@.

5.3.6 ODT Timeout - If the user specifies a nonexistent address or causes a parity error, console ODT responds to the error by printing ?<CR><LF>@.

5.3.7 Invalid Characters - Console ODT will recognize upper- or lowercase characters as commands. Any character that console ODT does not recognize during a particular sequence is echoed (except for ASCII characters in the range 0 - 17 (octal)), and console ODT prints ?<CR><LF>@.

5.4 DCJ11 PIPELINE PROCESSING

The DCJ11 gets much of its performance from its prefetch and predecode mechanisms. The primary benefit of prefetch and predecode is that memory references are overlapped with internal operations, and the need for explicit instruction fetch and decode cycles is minimized. The prefetch and predecode operations are performed automatically by the DCJ11 chip and cannot be altered by the user.

A primary function of the prefetch mechanism is to fill four registers with information and replenish the registers as required. These four registers, the virtual program counter (VPC), the physical program counter (PPC), the prefetch buffer (PB), and the instruction register (IR) are collectively referred to as the prefetch pipeline. The contents of registers in the beginning of the pipeline are used to determine the contents of registers further down the pipeline. When the pipeline is filled, the prefetch mechanism is said to be in steady state. Four microcycles are required to fill an empty pipeline. Figure 5-11 illustrates the process of filling the pipeline.

Microcycle 1	Microcycle 2	Microcycle 3	Microcycle 4
VPC <-- PC	PPC <-- MMU(VPC)	PB <-- M[PPC]	IR <-- PB
	VPC <-- VPC + 2	PPC <-- MMU(VPC)	PB <-- M[PPC]
		VPC <-- VPC + 2	PPC <-- MMU(VPC)
			VPC <-- VPC + 2
			PC <-- PC + 2
			MMR2 <-- PC

Figure 5-11 Pipeline Filling Process

In microcycle 1, the VPC is is simply set to the same value as the PC. In microcycle 2, the VPC is sent through the MMU and the resulting physical address is loaded into the PPC. The VPC is then incremented by 2. At this point we have a valid VPC and PPC and the pipeline is said to be synchronized. Sometimes while executing a macroinstruction, the pipeline is synchronized but not filled. In that case, only microcycles 3 and 4 need be performed for the next macroinstruction.

In microcycle 3, the word in memory addressed by the PPC is fetched into the PB. The PPC is updated with the relocated (mapped) VPC and the VPC is incremented again. In microcycle 4, the PB is sent to the IR and is decoded as the next macroinstruction (note that the DCJ11 asserts PDRC at this time). The new contents of the PB are fetched from the memory location referenced by the PPC. The PPC is again updated with the relocated (mapped) VPC and the VPC is updated (incremented) once again. Also during microcycle 4, the original PC is loaded into MMR2 (if MMR0<15:13> = 000) and is incremented by 2.

In steady state (i.e., when microcycle 4 is complete), the IR contains the macroinstruction being executed, the PB contains the data from the memory location pointed to by the PC, the PPC contains the physical address of the next word to be prefetched,

and the VPC contains the incremented value of the PC.

Once in steady state, a stream of macroinstructions that operate only on registers may be executed at the rate of one per microcycle (i.e., microcycle 4). While one instruction is being executed, the next one is being decoded, and the following one is being prefetched into the PB. As illustrated in Figure 5-11 during microcycle 4: the contents of the prefetch buffer are loaded into the IR, the word addressed by the PPC is loaded into the PB, the VPC is relocated and loaded into the PPC, and the VPC is incremented by 2. This maintains the steady state, allowing the next macroinstruction to be executed in the next microcycle. Note also that the DCJ11 bus is kept busy 100% of the time.

The instructions that operate on immediate data and a register also make maximum use of the prefetch mechanism. At steady state, a stream of these macroinstructions execute in two microcycles (microcycles 3 and 4). During microcycle 3, the data in the PB is moved to a scratch register. During microcycle 4, the operation is performed. In both cycles, the steady state of the prefetch mechanism is maintained by prefetching the next instruction stream word. The DCJ11 bus is again kept busy 100% of the time.

The prefetch pipeline is refilled after a power-up sequence or if a prefetch fault occurs. Prefetch faults occur when the PS, CCR, PC, or any of the memory management registers are written. A prefetch fault invalidates only the PB. This means that the pipeline remains synchronized and can be refilled in two microcycles.

5.4.1 Pipeline Flow Example - Consider the following example program:

Virtual Address	Symbolic Representation	Octal Code
1000	MOV R2,R3	010203
1002	BIS #1,R3	052703
		000001
1004	ADD R1,R3	060105
1006	CLR R0	005000
1012	ADD R3,R0	060300

The flow of information through the pipeline occurs as shown in Table 5-3.

Table 5-3 Pipeline Flow

Pipeline Register	Microcycle					
	n	n+1	n+2	n+3	n+4	n+5
PC	1002	1004	1006	1010	1012	1014
IR	MOV (010203)	BIS (052703)	BIS (052703)	ADD (060105)	CLR (005000)	ADD (060300)
PB	BIS (052703)	000001	ADD (060105)	CLR (005000)	ADD (060300)	*
PPC	MMU(1004)	MMU(1006)	MMU(1010)	MMU(1012)	MMU(1014)	MMU(1016)
VPC	1006	1010	1012	1014	1016	1020

* Instruction at location 1014

Note that the example starts at microcycle n, by which time the prefetch pipeline has been filled (i.e., the pipeline is in steady state). All the instructions in the example execute in one microcycle except the BIS instruction, which executes in two microcycles.

6.1 INTRODUCTION

The first part of this chapter is divided into six major sections:

- Single-Operand Addressing -- One part of the instruction word specifies the registers; the other part provides information for locating the operand.
- Double-Operand Addressing -- One part of the instruction word specifies the registers; the remaining parts provide information for locating two operands.
- Direct Addressing -- The operand is the content of the selected register.
- Deferred (Indirect) Addressing -- The contents of the selected register is the address of the operand.
- Use of the PC as a General-Purpose Register -- The PC is different from other general-purpose registers in one important respect. Whenever the processor retrieves an instruction, it automatically advances the PC by 2. By combining this automatic advancement of the PC with four of the basic addressing modes, we produce the four special PC modes -- immediate, absolute, relative, and relative-deferred.
- Use of the Stack Pointer as a General-Purpose Register -- General-purpose registers can be used for stack operations.

The second part of this chapter describes each of the instructions in the DCJ11 instruction set.

6.2 ADDRESSING MODES

Data stored in memory must be accessed and manipulated. Data handling is specified by a DCJ11 instruction (MOV, ADD, etc.), which usually specifies the:

- Function to be performed (operation code).
- General-purpose register to be used when locating the source operand, and/or destination operand (where required).
- Addressing mode, which specifies how the selected registers are to be used.

A large portion of the data handled by a computer is structured

(in character strings, arrays, lists, etc.). The DCJ11 addressing modes provide for efficient and flexible handling of structured data.

A general-purpose register may be used with an instruction in any of the following ways.

1. As an accumulator -- The data to be manipulated resides in the register.

2. As a pointer -- The contents of the register is the address of an operand, rather than the operand itself.

3. As a pointer that automatically steps through memory locations -- Automatically stepping forward through consecutive locations is known as autoincrement addressing; automatically stepping backwards is known as autodecrement addressing. These modes are particularly useful for processing tabular or array data.

4. As an index register -- In this instance, the contents of the register and the word following the instruction are summed to produce the address of the operand. This allows easy access to variable entries in a list.

An important DCJ11 feature, which should be considered with the addressing modes, is the register arrangement.

- Two sets of six general-purpose registers (R0--R5 and R0'--R5')
- A hardware stack pointer (SP) register (R6) for each processor mode (kernel, supervisor, user)
- A program counter (PC) register (R7)

Registers R0--R5 and R0'--R5' are not dedicated to any specific function; their use is determined by the instruction that is decoded.

- They can be used for operand storage. For example, the contents of two registers can be added and stored in another register.
- They can contain the address of an operand or serve as pointers to the address of an operand.
- They can be used for the autoincrement or autodecrement features.
- They can be used as index registers for convenient data and program access.

The DCJ11 also has instruction addressing mode combinations that facilitate temporary data storage structures. These can be used for convenient handling of data that must be accessed frequently. This is known as stack manipulation. The register that keeps track of stack manipulation is known as the stack pointer (SP).

Any register can be used as a stack pointer under program control; however, certain instructions associated with subroutine linkage and interrupt service automatically use register R6 as a "hardware stack pointer." For this reason, R6 is frequently referred to as the SP.

- The stack pointer (SP) keeps track of the latest entry on the stack.
- The stack pointer moves down as items are added to the stack and moves up as items are removed. Therefore, the stack pointer always points to the top of the stack.
- The hardware stack is used during trap or interrupt handling to store information, allowing an orderly return to the interrupted program.

Register R7 is used by the processor as its program counter (PC). It is recommended that R7 not be used as a stack pointer or accumulator. Whenever an instruction is fetched from memory, the program counter is automatically incremented by two to point to the next instruction word.

6.2.1 Single-Operand Addressing - The instruction format for all single-operand instructions (such as CLR, INC, TST) is shown in Figure 6-1.

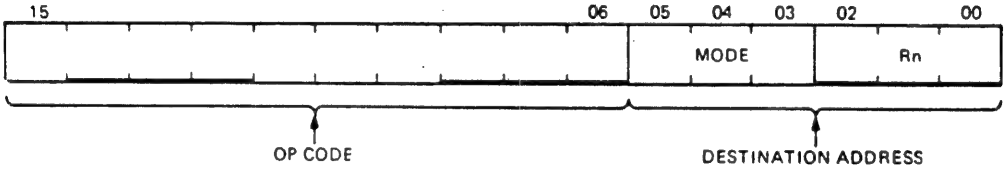


Figure 6-1 Single-Operand Addressing

Bits <15:6> specify the operation code that defines the type of instruction to be executed.

Bits <5:0> form a 6-bit field called the destination address field. The destination address field consists of two subfields:

- Bits <5:3> specify the destination mode. Bit 3 is set to indicate deferred (indirect) addressing.
- Bits <2:0> specify which of the 8 general-purpose registers is to be referenced by this instruction word.

6.2.2 Double-Operand Addressing - Operations that imply two operands (such as ADD, SUB, MOV, and CMP) are handled by instructions that specify two addresses. The first operand is called the source operand; the second is called the destination operand. Bit assignments in the source and destination address fields may specify different modes and different registers. The instruction format for the double operand instruction is shown in

Figure 6-2.

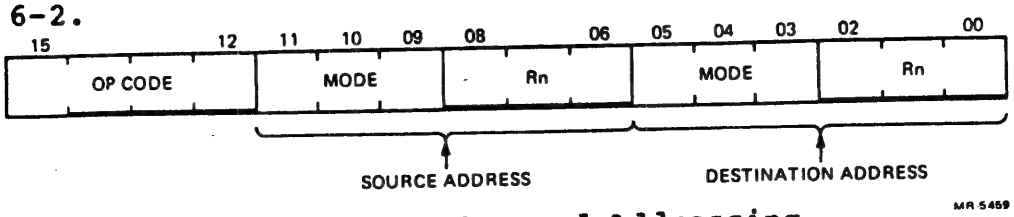


Figure 6-2 Double-Operand Addressing

The source address field is used to select the source operand (the first operand). The destination is used similarly, and locates the second operand and the result. For example, the instruction ADD A, B adds the contents (source operand) of location A to the contents (destination operand) of location B. After execution, B will contain the result of the addition and the contents of A will be unchanged.

Examples in this paragraph and the rest of the chapter use the following sample DCJ11 instructions. (A complete listing of the DCJ11 instructions appears in Paragraph 6.3.)

Mnemonic	Description	Octal Code
CLR	Clear. (Zero the specified destination.)	0050DD
CLRB	Clear byte. (Zero the byte in the specified destination.)	1050DD
INC	Increment. (Add one to contents of the destination.)	0052DD
INCB	Increment byte. (Add one to the contents of the destination byte.)	1052DD
COM	Complement. (Replace the contents of the destination by its logical complement; each 0 bit is set and each one bit is cleared.)	0051DD
COMB	Complement byte. (Replace the contents of the destination byte by its logical complement; each 0 bit is set and each 1 bit is cleared.)	1051DD
ADD	Add. (Add the source operand to the destination operand and store the result at the destination address.)	06SSDD

DD = destination field (six bits)
SS = source field (six bits)
() = contents of

6.2.3 Direct Addressing - The following summarizes the four basic modes used with direct addressing.

Direct Modes (Figures 6-3 to 6-6)

Mode	Name	Assembler Syntax	Function
0	Register	Rn	Register contains operand.

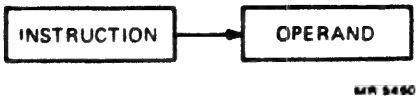


Figure 6-3 Mode 0 Register

Mode	Name	Assembler Syntax	Function
2	Autoincrement	(Rn)+	Register is used as a pointer to sequential data and then incremented.

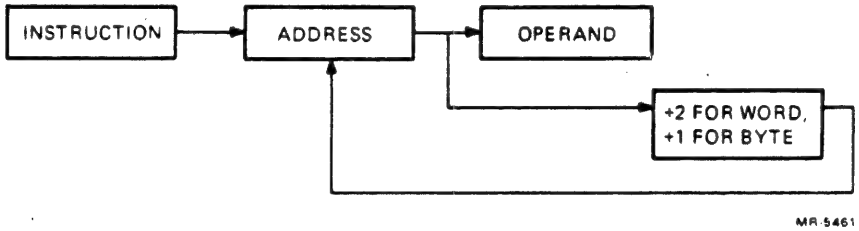


Figure 6-4 Mode 2 Autoincrement

Mode	Name	Assembler Syntax	Function
4	Autodecrement	-(Rn)	Register is decremented and then used as a pointer.

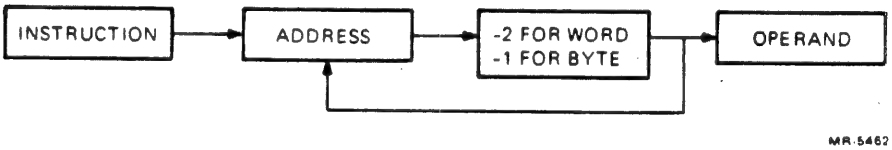


Figure 6-5 Mode 4 Autodecrement

Mode	Name	Assembler Syntax	Function
6	Index	X(Rn)	Value X is added to (Rn) to produce address of operand. Neither X nor (Rn) is modified.

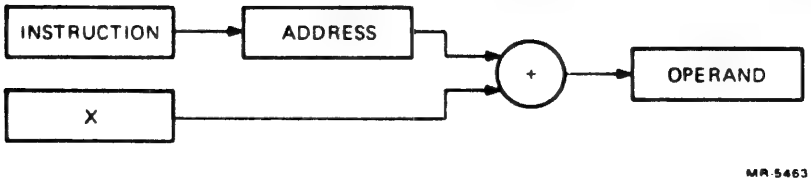


Figure 6-6 Mode 6 Index

6.2.3.1 Register Mode - With register mode any of the general registers may be used as simple accumulators, with the operand contained in the selected register. Since they are hardware registers (within the processor), the general registers operate at high speeds and provide speed advantages when used for operating on frequently accessed variables. The assembler interprets and assembles instructions of the form OPR Rn as register mode operations. Rn represents a general register name or number and OPR is used to represent a general instruction mnemonic. Assembler syntax requires that a general register be defined as follows.

```
R0 = %0 (% sign indicates register definition)
R1 = %1
R2 = %2, etc.
```

Registers are typically referred to by name as R0, R1, R2, R3, R4, R5, R6, and R7. However, R6 and R7 are also referred to as SP and PC, respectively.

OPR Rn

Register Mode Examples (Figures 6-7 to 6-9)

1.	Symbolic	Octal Code	Instruction Name
	INC R3	005203	Increment

Operation: Add one to the contents of general-purpose register R3.

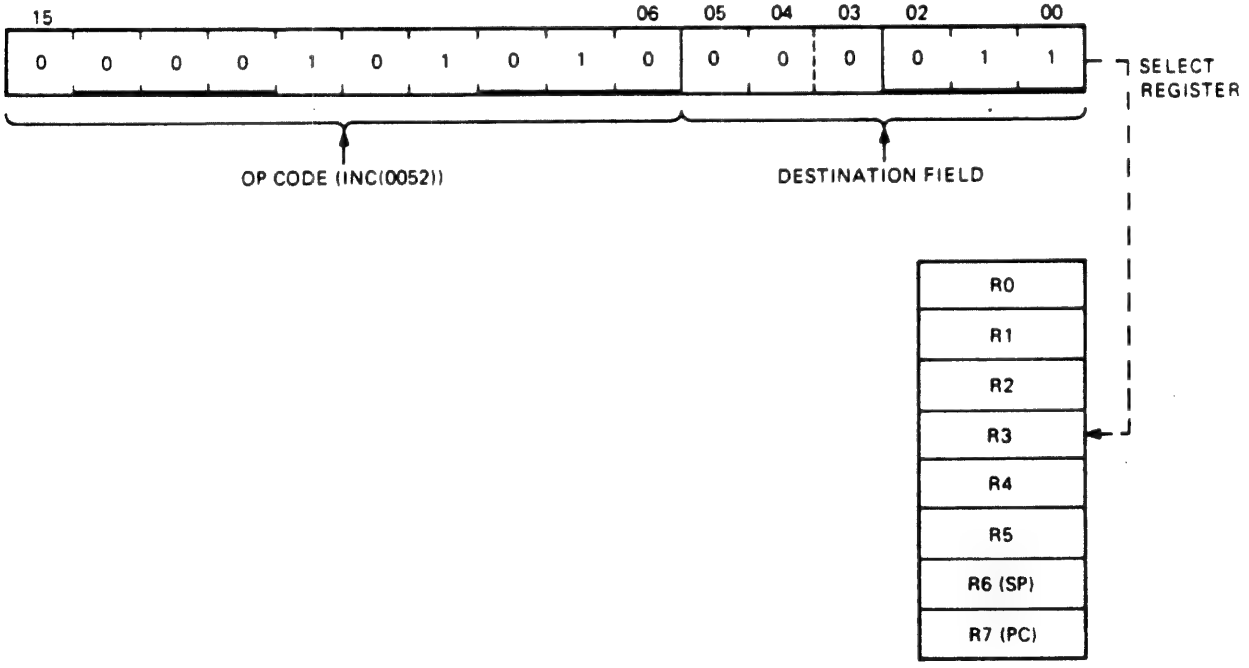


Figure 6-7 INC R3 Increment

2.	Symbolic	Octal Code	Instruction Name
	ADD R2, R4	060204	Add

Operation: Add the contents of R2 to the contents of R4.

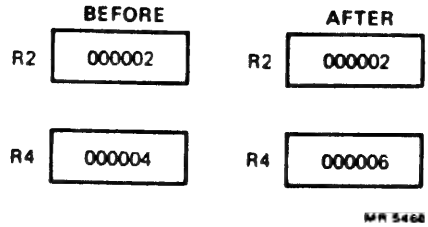


Figure 6-8 ADD R2,R4 Add

3. Symbolic Octal Code Instruction Name

COMB R4 105104 Complement byte

Operation: 1's complement bits <7:0> (byte) in R4. (When general registers are used, byte instructions operate only on bits <7:0>; i.e., byte 0 of the register.)

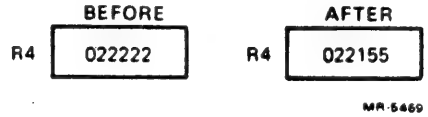


Figure 6-9 COMB R4 Complement Byte

6.2.3.2 Autoincrement Mode [OPR (Rn)+] - This mode (mode 2) provides for automatic stepping of a pointer through sequential elements of a table of operands. It assumes the contents of the selected general-purpose register to be the address of the operand. Contents of registers are stepped (by one for byte instructions, by two for word instructions, always by two for R6 and R7) to address the next sequential location. The autoincrement mode is especially useful for array processing and stack processing. It will access an element of a table and then step the pointer to address the next operand in the table. Although most useful for table handling, this mode is completely general and may be used for a variety of purposes.

OPR (Rn)+

Autoincrement Mode Examples (Figures 6-10 to 6-12)

1. Symbolic Octal Code Instruction Name

CLR (R5)+ 005025 Clear

Operation: Use contents of R5 as the address of the operand. Clear selected operand and then increment the contents of R5 by two.

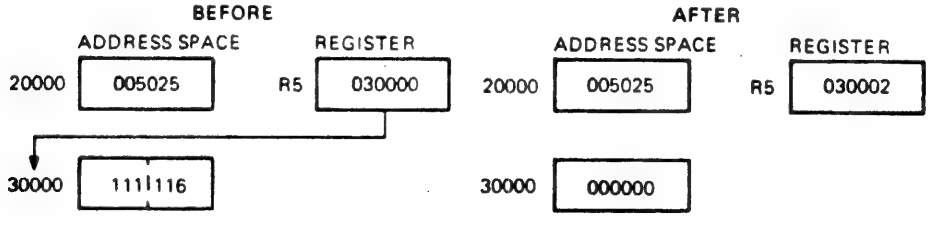


Figure 6-10 CLR (R5)+ Clear

2. Symbolic Octal Code Instruction Name

CLRB (R5)+ 105025 Clear byte

Operation: Use contents of R5 as the address of the operand. Clear selected byte operand and then increment the contents of R5 by one.

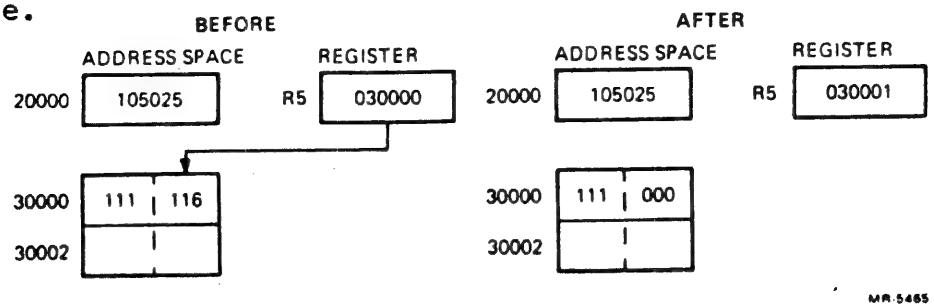


Figure 6-11 CLRB (R5)+ Clear Byte

3.	Symbolic	Octal Code	Instruction Name
	ADD (R2)+,R4	062204	Add

Operation: The contents of R2 are used as the address of the operand, which is added to the contents of R4. R2 is then incremented by two.

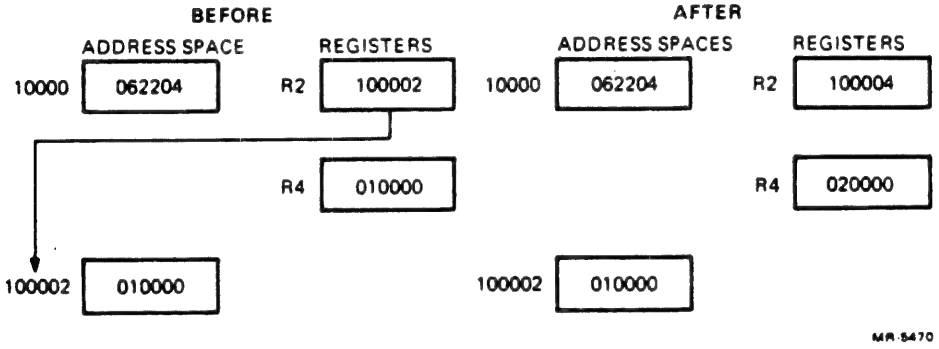


Figure 6-12 ADD (R2)+,R4 Add

6.2.3.3 Autodecrement Mode [OPR-(Rn)] - This mode (mode 4) is useful for processing data in a list in reverse direction. The contents of the selected general-purpose register are decremented (by one for byte instructions, by two for word instructions) and then used as the address of the operand. The choice of postincrement, predecrement features for the DCJ11 were not arbitrary decisions, but were intended to facilitate hardware/software stack operations.

OPR-(Rn)

Autodecrement Mode Examples (Figures 6-13 to 6-15)

1.	Symbolic	Octal Code	Instruction Name
	INC -(R0)	005240	Increment

Operation: The contents of R0 are decremented by two and used as the address of the operand. The operand is incremented by one.

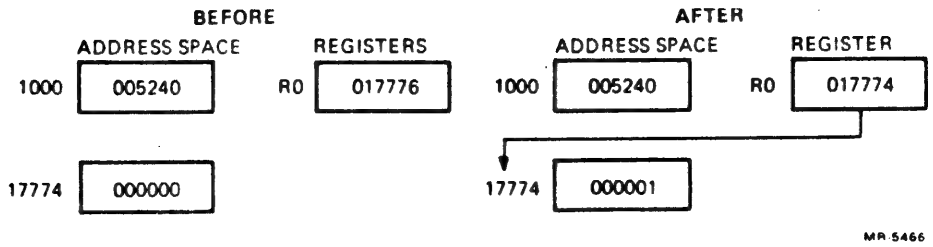


Figure 6-13 INC -(R0) Increment

2. Symbolic	Octal Code	Instruction Name
INCB -(R0)	105240	Increment byte

Operation: The contents of R0 are decremented by one and then used as the address of the operand. The operand byte is increased by one.

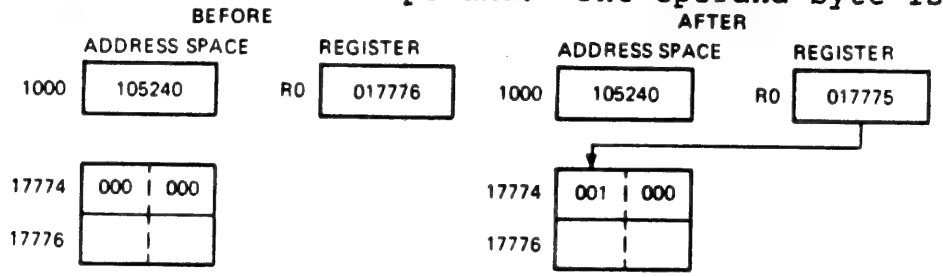


Figure 6-14 INCB -(R0) Increment Byte

3. Symbolic	Octal Code	Instruction Name
ADD -(R3),R0	064300	Add

Operation: The contents of R3 are decremented by two and then used as a pointer to an operand (source), which is added to the contents of R0 (destination operand).

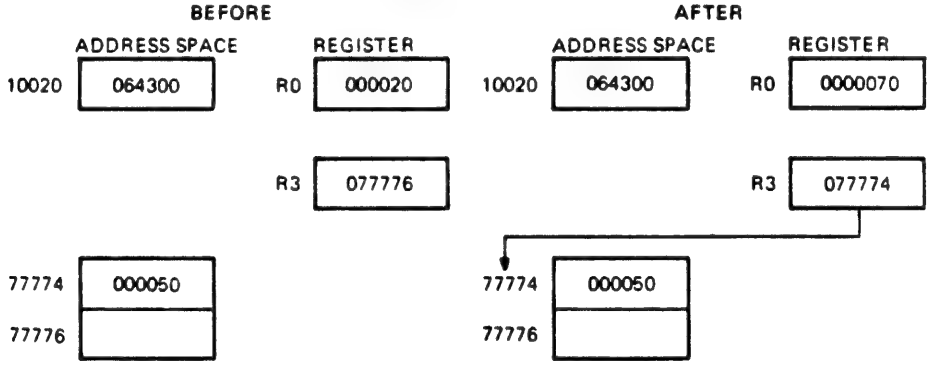


Figure 6-15 ADD -(R3),R0 Add

6.2.3.4 Index Mode [OPR X(Rn)] - In this mode (mode 6) the contents of the selected general-purpose register, and an index word following the instruction word, are summed to form the address of the operand. The contents of the selected register may be used as a base for calculating a series of addresses, thus allowing random access to elements of data structures. The selected register can then be modified by program to access data in the table. Index addressing instructions are of the form OPR

X(Rn), where X is the indexed word located in the memory location following the instruction word and Rn is the selected general-purpose register.

OPR X(Rn)

Index Mode Examples (Figures 6-16 to 6-18)

1.	Symbolic	Octal Code	Instruction Name
	CLR 200(R4)	005064 000200	Clear

Operation: The address of the operand is determined by adding 200 to the contents of R4. The operand location is then cleared.

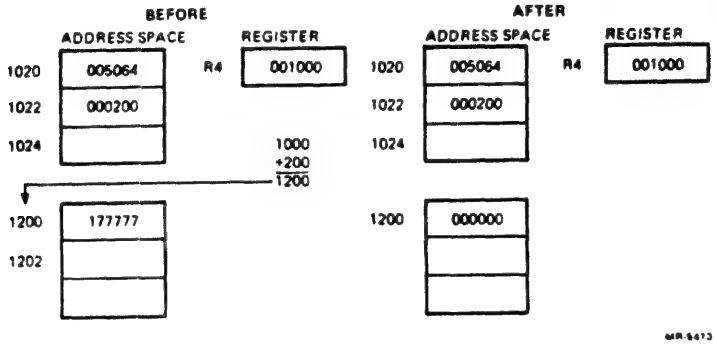


Figure 6-16 CLR 200(R4) Clear

2.	Symbolic	Octal Code	Instruction Name
	COMB 200(R1)	105161 000200	Complement byte

Operation: The contents of a location, which are determined by adding 200 to the contents of R1, are 1's complemented (i.e., logically complemented).

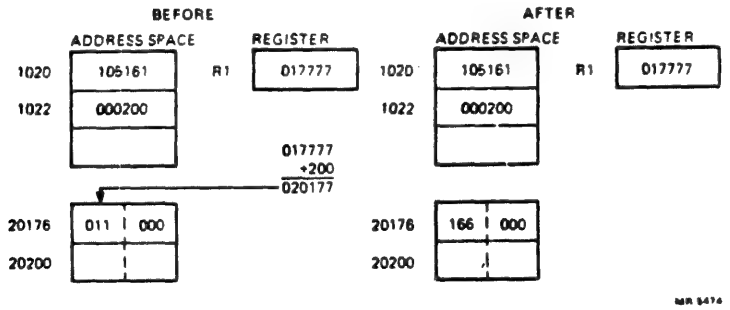


Figure 6-17 COMB 200(R1) Complement Byte

3.	Symbolic	Octal Code	Instruction Name
	ADD 30(R2),20(R5)	066265 000030 000020	Add

Operation: The contents of a location, which are determined by

adding 30 to the contents of R2, are added to the contents of a location that is determined by adding 20 to the contents of R5. The result is stored at the destination address, that is, 20(R5).

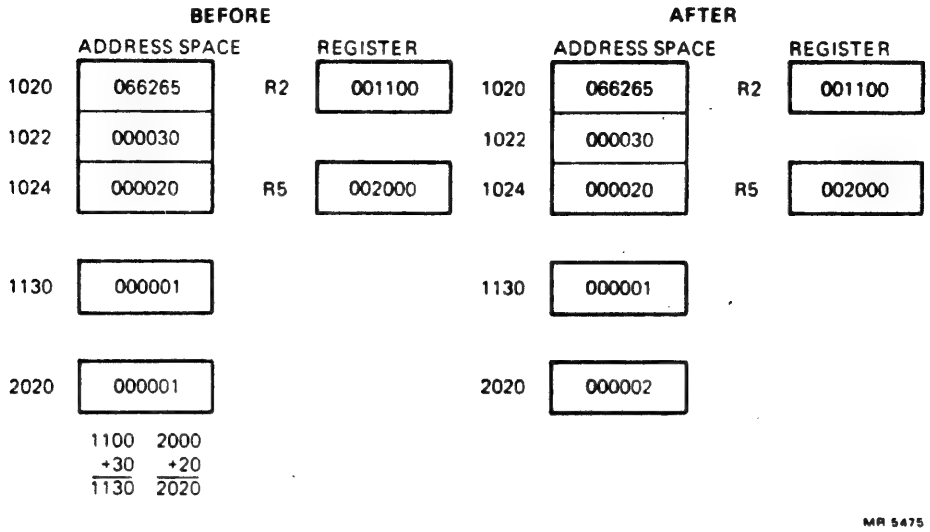


Figure 6-18 ADD 30(R2),20(R5) Add

6.2.4 Deferred (Indirect) Addressing - The four basic modes may also be used with deferred addressing. Whereas in register mode the operand is the contents of the selected register, in register-deferred mode the contents of the selected register is the address of the operand.

In the three other deferred modes, the contents of the register select the address of the operand rather than the operand itself. These modes are therefore used when a table consists of addresses rather than operands. The assembler syntax for indicating deferred addressing is @ [or () when this is not ambiguous]. The following summarizes the deferred versions of the basic modes.

Deferred Modes (Figures 6-19 to 6-22)

Mode	Name	Assembler Syntax	Function
1	Register-deferred	@Rn or (Rn)	Register contains the address of the operand.

Figure 6-19 Mode 1 Register-Deferred

Mode	Name	Assembler Syntax	Function
3	Autoincrement-Deferred	@(Rn)+	Register is first used as a

pointer to a word containing the address of the operand and then incremented (always by two, even for byte instructions).

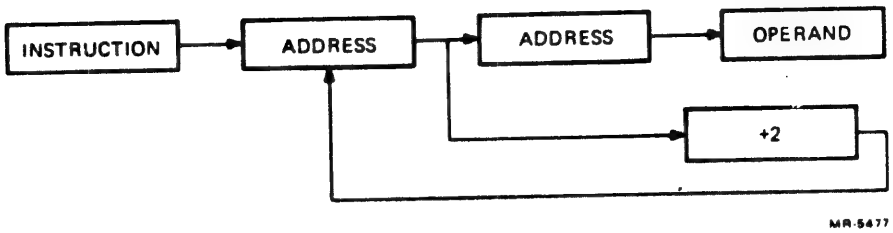


Figure 6-20 Mode 3 Autoincrement-Deferred

Mode	Name	Assembler Syntax	Function
5	Autodecrement-deferred	@-(Rn)	Register is decremented (always by two, even for byte instructions) and then used as a pointer to a word containing the address of the operand.

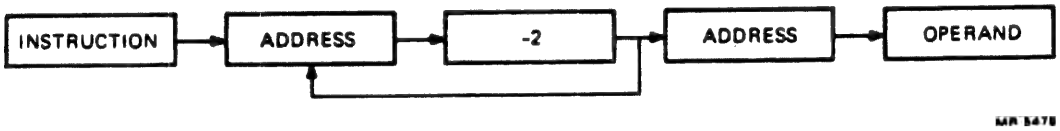


Figure 6-21 Mode 5 Autodecrement-Deferred

Mode	Name	Assembler Syntax	Function
7	Index-deferred	@X(Rn)	Value X (stored in a word following the instruction) and (Rn) are added; the sum is used as a pointer to a word containing the address of the operand. Neither X nor (Rn) is modified.

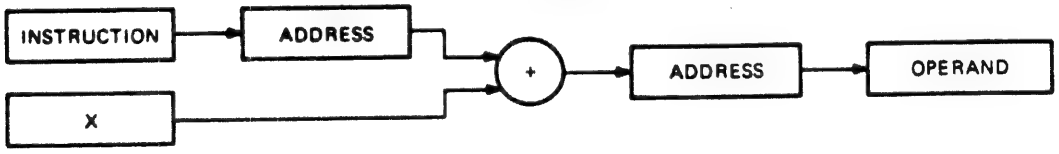


Figure 6-22 Mode 7 Index-Deferred

The following examples illustrate the deferred modes.

Register-Deferred Mode Example (Figure 6-23)

Symbolic	Octal Code	Instruction Name
CLR @R5	005015	Clear

Operation: The contents of location specified in R5 are cleared.

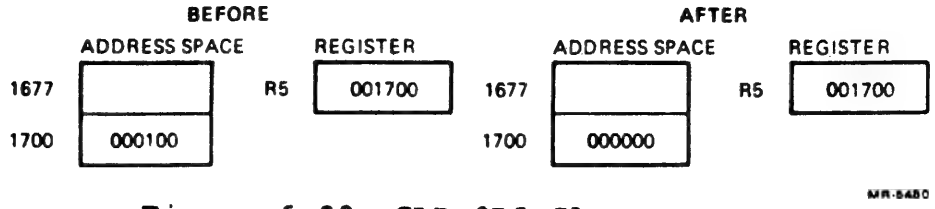


Figure 6-23 CLR @R5 Clear

Autoincrement-Deferred Mode Example (Mode 3) (Figure 6-24)

Symbolic	Octal Code	Instruction Name
INC @(R2)+	005232	Increment

Operation: The contents of R2 are used as the address of the address of the operand. The operand is increased by one; the contents of R2 are incremented by two.

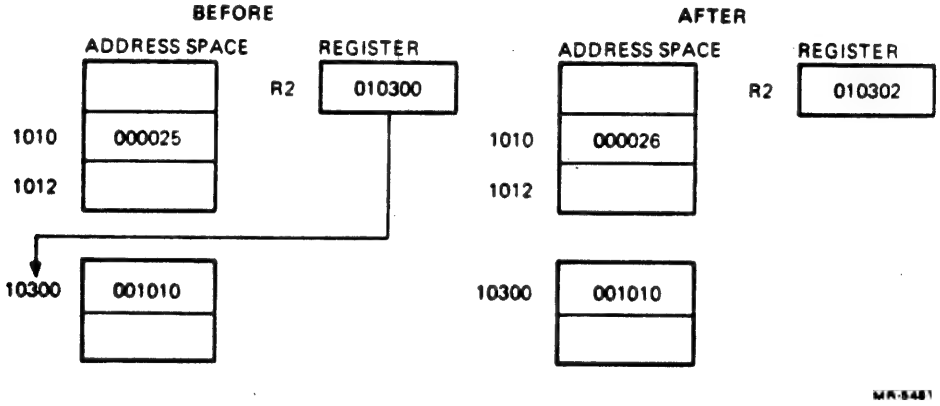


Figure 6-24 INC @(R2)+ Increment

Autodecrement-Deferred Mode Example (Mode 5) (Figure 6-25)

Symbolic	Octal Code
COM @-(R0)	005150

Operation: The contents of R0 are decremented by two and then used as the address of the address of the operand. The operand is 1's complemented (i.e., logically complemented).

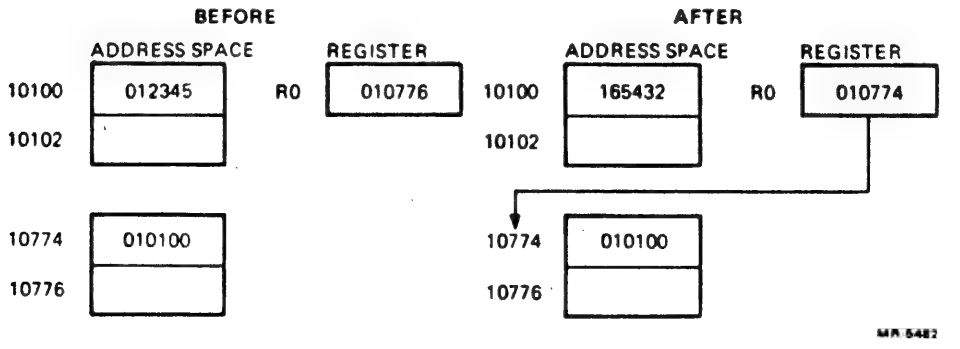


Figure 6-25 COM @-(R0) Complement

Index-Deferred Mode Example (Mode 7) (Figure 6-26)

Symbolic	Octal Code	Instruction Name
ADD @1000(R2),R1	067201 001000	Add

Operation: 1000 and the contents of R2 are summed to produce the address of the address of the source operand, the contents of which are added to the contents of R1; the result is stored in R1.

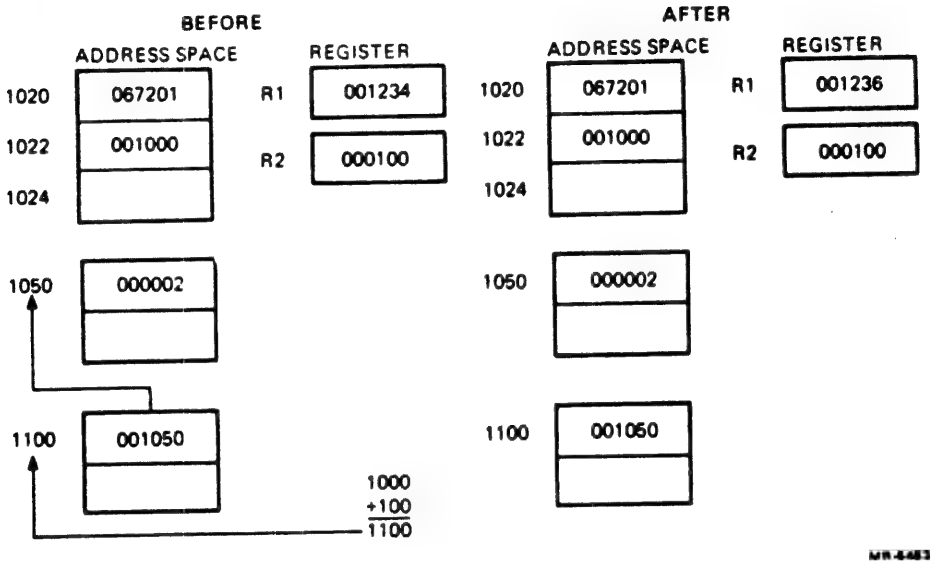


Figure 6-26 ADD @1000(R2),R1 Add

6.2.5 Use Of The PC As A General-Purpose Register - Although register 7 is a general-purpose register, it doubles in function as the program counter for the DCJ11. Whenever the processor uses the program counter to acquire a word from memory, the program counter is automatically incremented by two to contain the address of the next word of the instruction being executed or the address of the next instruction to be executed. (When the program uses the PC to locate byte data, the PC is still incremented by two.)

The PC responds to all the standard DCJ11 addressing modes. However, with four of these modes the PC can provide advantages for handling position-independent code and unstructured data. When utilizing the PC, these modes are termed immediate, absolute (or immediate-deferred), relative, and relative-deferred. The modes are summarized below.

Mode	Name	Assembler Syntax	Function
2	Immediate	#n	Operand follows instruction.
3	Absolute	@#A	Absolute address of operand follows instruction.

6	Relative	A	Relative address (index value) follows the instruction.
7	Relative-deferred	@A	Index value (stored in the word after the instruction) is the relative address for the address of the operand.

When a standard program is available for different users, it is often helpful to be able to load it into different areas of memory and run it in those areas. The DCJ11 can accomplish the relocation of a program very efficiently through the use of position-independent code (PIC), which is written by using the PC addressing modes. If an instruction and its operands are moved in such a way that the relative distance between them is not altered, the same offset relative to the PC can be used in all positions in memory. Thus, PIC usually references locations relative to the current location.

The PC also greatly facilitates the handling of unstructured data. This is particularly true of the immediate and relative modes.

6.2.5.1 Immediate Mode [OPR N,DD] - Immediate mode (mode 2) is equivalent in use to the autoincrement mode with the PC. It provides time improvements for accessing constant operands by including the constant in the memory location immediately following the instruction word.

OPR #n,DD

Immediate Mode Example (Figure 6-27)

Symbolic	Octal Code	Instruction Name
ADD #10,R0	062700 000010	Add

Operation: The value 10 is located in the second word of the instruction and is added to the contents of R0. Just before this instruction is fetched and executed, the PC points to the first word of the instruction. The processor fetches the first word and increments the PC by two. The source operand mode is 27 (autoincrement the PC). Thus, the PC is used as a pointer to fetch the operand (the second word of the instruction) before it is incremented by two to point to the next instruction.

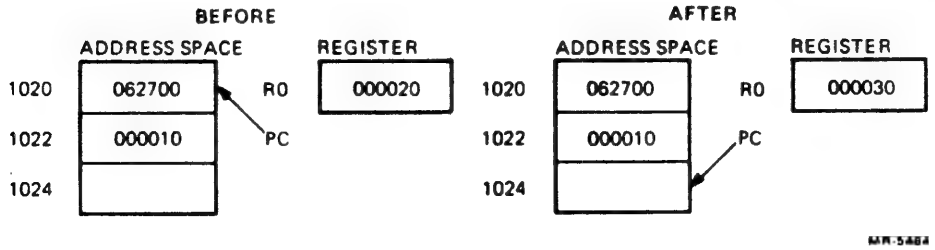


Figure 6-27 ADD #10,R0 Add

6.2.5.2 Absolute Addressing [OPR @ A] - This mode (mode 3) is the equivalent of immediate-deferred or autoincrement-deferred using the PC. The contents of the location following the instruction are taken as the address of the operand. Immediate data is interpreted as an absolute address (i.e., an address that remains constant no matter where in memory the assembled instruction is executed).

OPR @#A

Absolute Mode Examples (Figures 6-28 and 6-29)

1.	Symbolic	Octal Code	Instruction Name
	CLR @#1100	005037 001100	Clear

Operation: Clear the contents of location 1100.

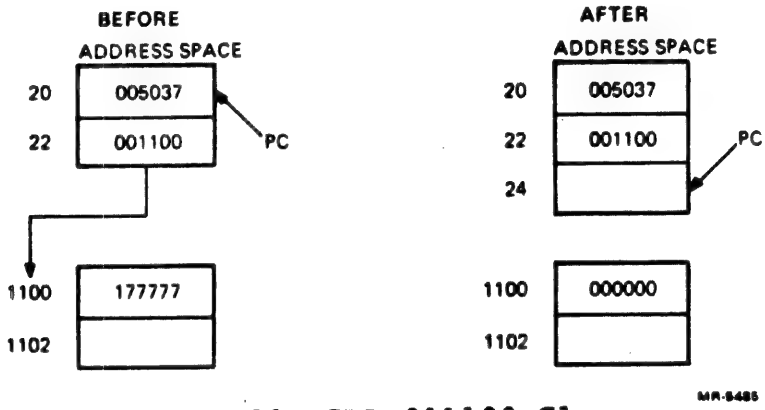


Figure 6-28 CLR @#1100 Clear

2.	Symbolic	Octal Code	Instruction Name
	ADD @#2000,R3	063703 002000	Add

Operation: Add contents of location 2000 to R3.

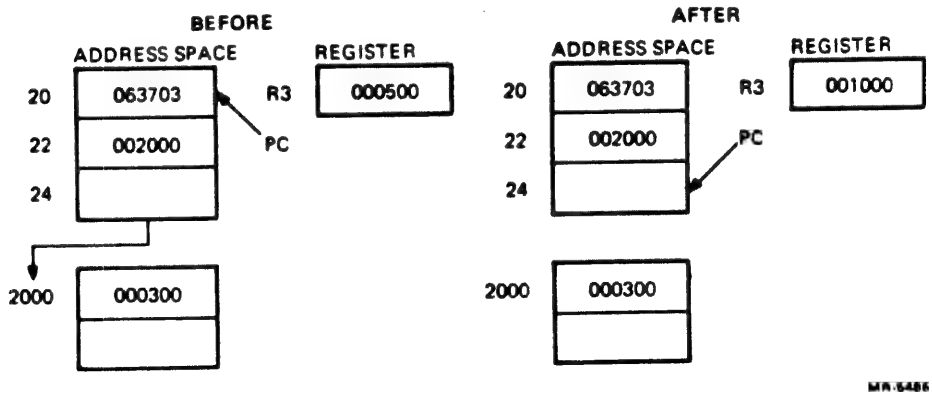


Figure 6-29 ADD @#2000 Add

6.2.5.3 Relative Addressing [OPR A Or OPR X(PC)] - This mode (mode 6) is assembled as index mode using R7. The base of the address calculation, which is stored in the second or third word of the instruction, is not the address of the operand, but the number which, when added to the (PC), becomes the address of the operand. This mode is useful for writing position-independent code since the location referenced is always fixed relative to the PC. When instructions are to be relocated, the operand is moved by the same amount.

OPR A or OPR X(PC) (X is the location of A relative to the instruction)

Relative Addressing Example (Figure 6-30)

Symbolic	Octal Code	Instruction Name
INC A	005267 000054	Increment

Operation: To increment location A, contents of memory location immediately following instruction word are added to (PC) to produce address A. Contents of A are increased by one.

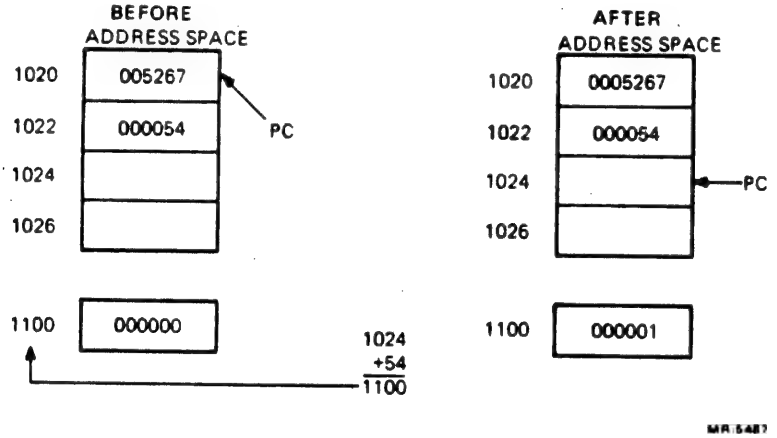


Figure 6-30 INC A Increment

6.2.5.4 Relative-Deferred Addressing [OPR @A Or OPR @X(PC)] - This mode (mode 7) is similar to relative mode, except that the second word of the instruction, when added to the PC, contains the address of the address of the operand, rather than the address of the operand.

OPR @A or OPR @X(PC) (X is the location containing the address of A, relative to the instruction)

Relative-Deferred Mode Example (Figure 6-31)

Symbolic	Octal Code	Instruction Name
CLR @A	005077 000020	Clear

Operation: Add second word of instruction to updated PC to produce address of address of operand. Clear operand.

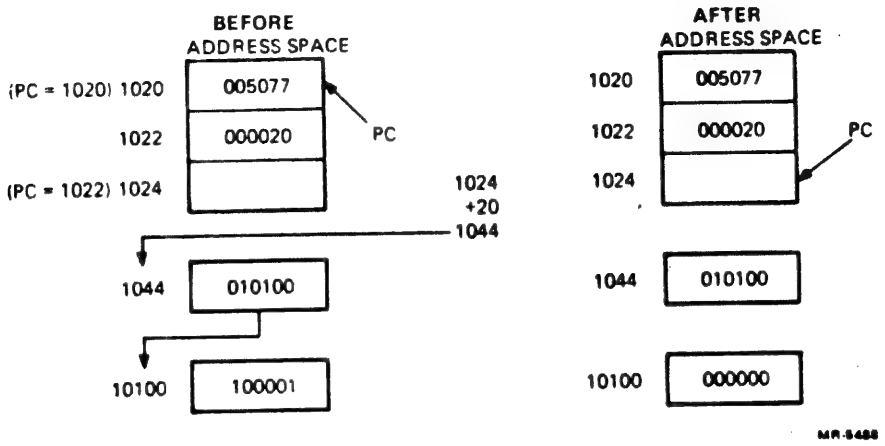


Figure 6-31 CLR @A Clear

6.2.6 Use Of The Stack Pointer As A General-Purpose Register - The processor stack pointer (SP, register 6) is in most cases the general register used for the stack operations related to program nesting. Autodecrement with register 6 "pushes" data onto the stack and autoincrement with register 6 "pops" data off the stack. Since the SP is used by the processor for interrupt handling, it has a special attribute: autoincrements and autodecrements are always done in steps of two. Byte operations using the SP in this way leave odd addresses unmodified.

6.3 INSTRUCTION SET

The rest of this chapter describes the DCJ11's instruction set. Each instruction's explanation includes the instruction's mnemonic, octal code, binary code, a diagram showing the format of the instruction, a symbolic notation describing its execution and effect on the condition codes, a description, special comments, and examples.

Each instruction's explanation is headed by its mnemonic. When the word instruction has a byte equivalent, the byte mnemonic also appears.

The diagram that accompanies each instruction shows the octal op code, binary op code, and bit assignments. [Note that in byte instructions the most significant bit (bit 15) is always a one.]

Symbols:

() = contents of

SS or src = source address

DD or dst = destination address

loc = location

<-- = becomes

↑ = "is popped from stack"

↓ = "is pushed onto stack"

∧ = boolean AND

∨ = boolean OR

⊻ = exclusive OR

~ = boolean not

REG or R = register

B = Byte

■ = 0 for word, 1 for byte

, = concatenated

6.3.1 Instruction Formats - The following formats include all instructions used in the DCJ11. Refer to individual instructions for more detailed information.

1. Single-Operand Group: (Figure 6-32)
   CLR, CLRB, COM, COMB, INC, INCB, DEC, DECB, NEG, NEGB, ADC, ADCB, SBC, SBCB, TST, TSTB, ROR, RORB, ROL, ROLB, ASR, ASRB, ASL, ASLB, JMP, SWAB, MFPS, MTPS, SXT, TSTSET, WRTLCK, XOR

Figure 6-32 Single-Operand Group

2. Double-Operand Group:

   a. Group 1: (Figure 6-33)
      BIT, BITB, BIC, BICB, BIS, BISB, ADD, SUB, MOV, MOVB, CMP, CMPB

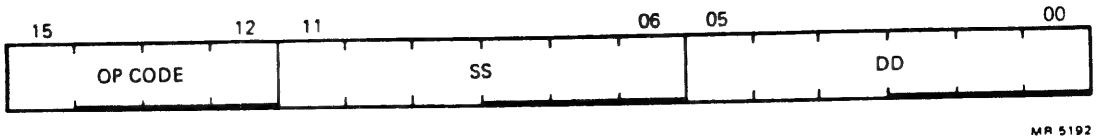


Figure 6-33 Double-Operand Group 1

b. Group 2: ASH, ASHC, DIV, MUL
(Figure 6-34)

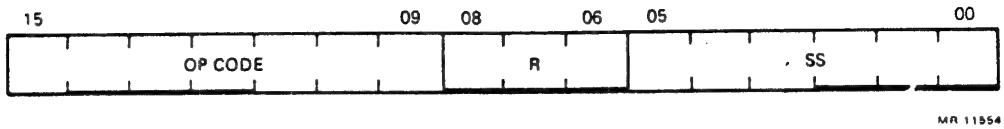


Figure 6-34 Double-Operand Group 2

3. Program Control Group:

a. Branch (all branch instructions) (Figure 6-35)

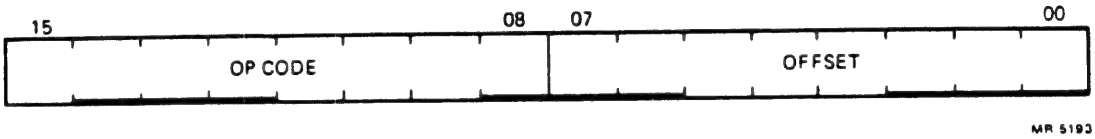


Figure 6-35 Program Control Group Branch

b. Jump to Subroutine (JSR) (Figure 6-36)

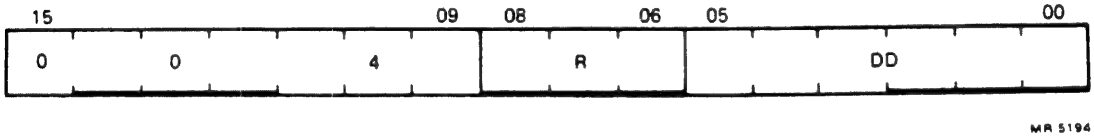


Figure 6-36 Program Control Group JSR

c. Subroutine Return (RTS) (Figure 6-37)

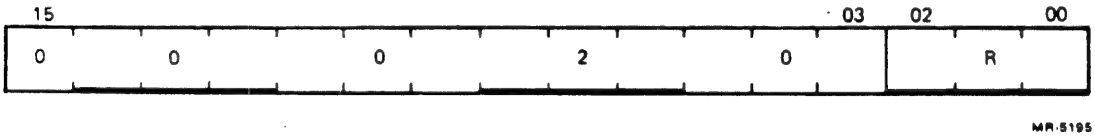


Figure 6-37 Program Control Group RTS

d. Traps (breakpoint, IOT, EMT, TRAP, BPT) (Figure 6-38)

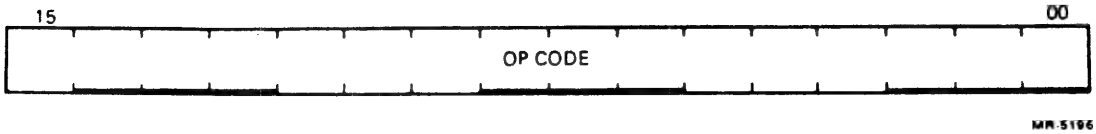


Figure 6-38 Program Control Group Traps

e. Subtract 1 and Branch (if = 0) (SOB) (Figure 6-39)

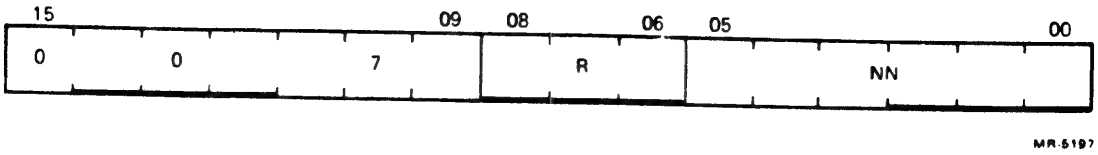


Figure 6-39 Program Control Group Subtract

f. Mark (Figure 6-40)

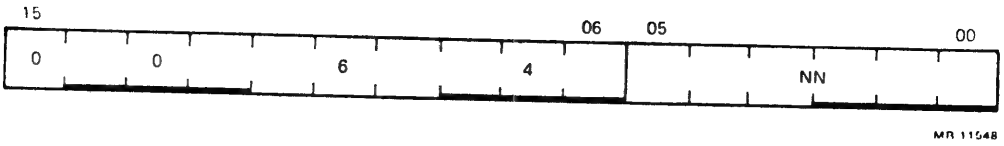


Figure 6-40 Mark

g. Call to Supervisor Mode (CSM) (Figure 6-41)

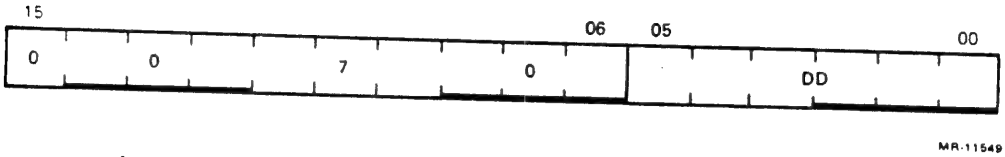


Figure 6-41 Call to Supervisor Mode

h. Set Priority Level (SPL) (Figure 6-42)

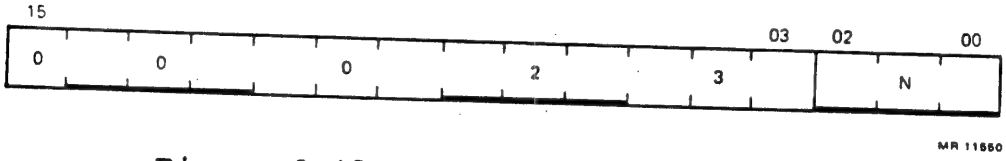


Figure 6-42 Set Priority Level

4. Operate Group: (Figure 6-43) HALT, WAIT, RTI, RESET, RTT, NOP, MFPT

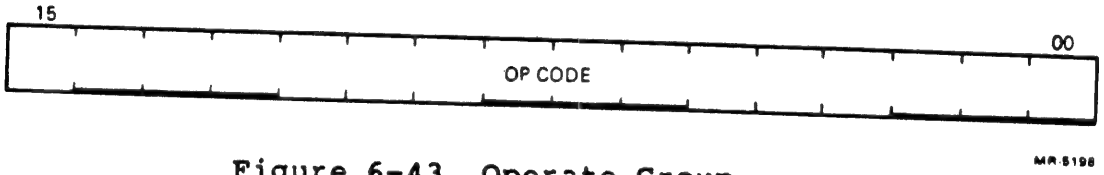


Figure 6-43 Operate Group

5. Condition Code Operators (all condition code instructions) (Figure 6-44)

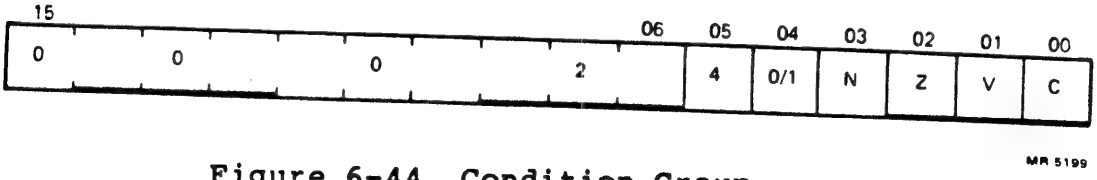


Figure 6-44 Condition Group

6. Move To/From Previous Instruction/Data Space Group: MTPD, MTPI, MFPD, MFPI

(Figure 6-45)

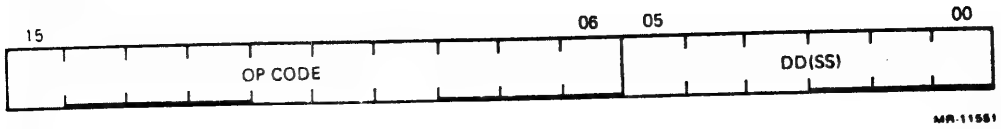


Figure 6-45 Move To And From Previous Instruction/Data Space Group

6.3.2 Byte Instructions - The DCJ11 includes a full complement of instructions that manipulate byte operands. Since all DCJ11 addressing is byte-oriented, byte manipulation addressing is straightforward. Byte instructions with autoincrement or autodecrement direct addressing cause the specified register to be modified by one to point to the next byte of data. Byte operations in register mode access the low-order byte of the specified register. These provisions enable the DCJ11 to perform as either a word or byte processor. The numbering scheme for word and byte addresses in memory is shown in Figure 6-46.

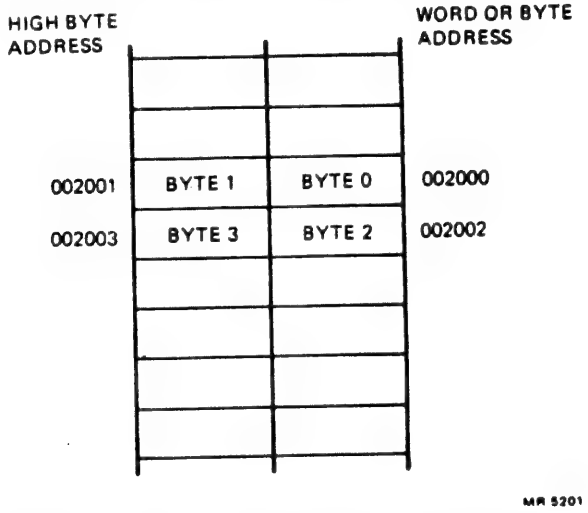


Figure 6-46 Byte Instructions

The most significant bit (bit 15) of the instruction word is set to indicate a byte instruction.

Example:

Symbolic	Octal Code	Instruction Name
CLR	0050DD	Clear word
CLRB	1050DD	Clear byte

6.3.3 List Of Instructions - The following is a list of the DCJ11 instruction set.

SINGLE-OPERAND

General

Mnemonic	Instruction	Op Code
CLR(B)	Clear destination	■050DD
COM(B)	Complement destination	■051DD
INC(B)	Increment destination	■052DD
DEC(B)	Decrement destination	■053DD
NEG(B)	Negate destination	■054DD
TST(B)	Test destination	■057DD
WRTLCK	Read/lock destination, write/unlock R0 into destination	0073DD
TSTSET	Test destination, set low bit	0072DD

Shift and Rotate

Mnemonic	Instruction	Op Code
ASR(B)	Arithmetic shift right	■062DD
ASL(B)	Arithmetic shift left	■063DD
ROR(B)	Rotate right	■060DD
ROL(B)	Rotate left	■061DD
SWAB	Swap bytes	0003DD

Multiple-Precision

Mnemonic	Instruction	Op Code
ADC(B)	Add carry	■055DD
SBC(B)	Subtract carry	■056DD
SXT	Sign extend	0067DD

PS Word Operators

Mnemonic	Instruction	Op Code
MFPS	Move byte from PS	1067DD
MTPS	Move byte to PS	1064SS

DOUBLE-OPERAND

General

Mnemonic	Instruction	Op Code
MOV(B)	Move source to destination	■1SSDD
CMP(B)	Compare source to destination	■2SSDD
ADD	Add source to destination	06SSDD

SUB	Subtract source from destination	16SSDD
ASH	Arithmetic shift	072RSS
ASHC	Arithmetic shift combined	073RSS
MUL	Multiply	070RSS
DIV	Divide	071RSS

Logical

Mnemonic	Instruction	Op Code
BIT(B)	Bit test	■3SSDD
BIC(B)	Bit clear	■4SSDD
BIS(B)	Bit set	■5SSDD
XOR	Exclusive OR	074RDD

PROGRAM CONTROL

Mnemonic	Instruction	Op Code or Base Code

Branch

BR	Branch (unconditional)	000400
BNE	Branch if not equal (to zero)	001000
BEQ	Branch if equal (to zero)	001400
BPL	Branch if plus	100000
BMI	Branch if minus	100400
BVC	Branch if overflow is clear	102000
BVS	Branch if overflow is set	102400
BCC	Branch if carry is clear	103000
BCS	Branch if carry is set	103400

Signed Conditional Branch

Mnemonic	Instruction	Op Code or Base Code
BGE	Branch if greater than or equal (to zero)	002000
BLT	Branch if less than (zero)	002400
BGT	Branch if greater than (zero)	003000
BLE	Branch if less than or equal (to zero)	003400

Unsigned Conditional Branch

Mnemonic	Instruction	Op Code or Base Code
BHI	Branch if higher	101000
BLOS	Branch if lower or same	101400
BHIS	Branch if higher or same	103000
BLO	Branch if lower	103400

Jump and Subroutine

Mnemonic	Instruction	Op Code or Base Code
JMP	Jump	0001DD
JSR	Jump to subroutine	004RDD
RTS	Return from subroutine	00020R
SOB	Subtract one and branch (if ≠ 0)	077R00

Trap and Interrupt

Mnemonic	Instruction	Op Code or Base Code
EMT	Emulator trap	104000 - 104377
TRAP	Trap	104400 - 104777
BPT	Breakpoint trap	000003
IOT	Input/output trap	000004
RTI	Return from interrupt	000002
RTT	Return from interrupt	000006

Miscellaneous Program Control

Mnemonic	Instruction	Op Code or Base Code
CSM	Call to supervisor mode	0070DD
MARK	Mark	0064NN
SPL	Set Priority Level	00023N

MISCELLANEOUS

Mnemonic	Instruction	Op Code or Base Code
HALT	Halt	000000
WAIT	Wait for interrupt	000001
RESET	Reset external bus	000005
MFPT	Move processor type	000007
MTPD	Move to previous data space	1066SS
MTPI	Move to previous instruction space	0066SS
MFPD	Move from previous data space	0065SS
MFPI	Move from previous instruction space	1065SS

CONDITION CODE OPERATORS

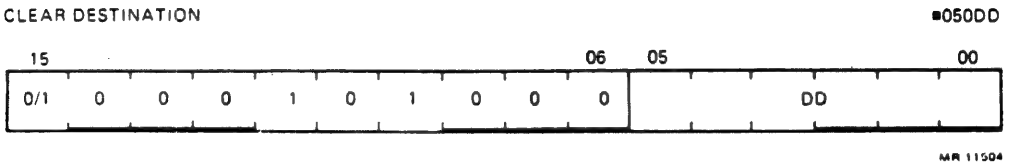
Mnemonic	Instruction	Op Code or Base Code
CLC	Clear C	000241
CLV	Clear V	000242
CLZ	Clear Z	000244
CLN	Clear N	000250

CCC	Clear all CC bits	000257
SEC	Set C	000261
SEV	Set V	000262
SEZ	Set Z	000264
SEN	Set N	000270
SCC	Set all CC bits	000277
NOP	No operation	000240

6.3.4 Single-Operand Instructions - The DCJ11 instructions that involve only one operand are described in the paragraphs that follow.

6.3.4.1 General -

CLR
CLRB



Operation: (dst) <-- 0

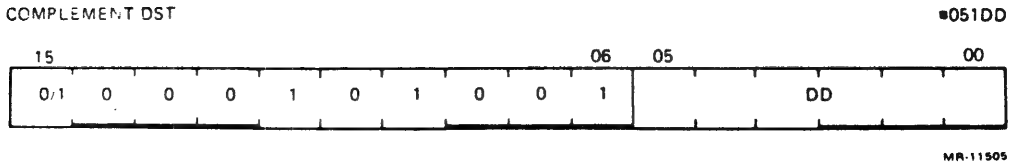
Condition Codes: N: cleared
Z: set
V: cleared
C: cleared

Description: Word: The contents of the specified destination are replaced with 0s.
Byte: Same.

Example: CLR R1

Before	After
(R1) = 177777	(R1) = 000000
NZVC	NZVC
1111	0100

COM
COMB



Operation: (dst) <-- ~ (dst)

Condition Codes: N: set if most significant bit of result is set;

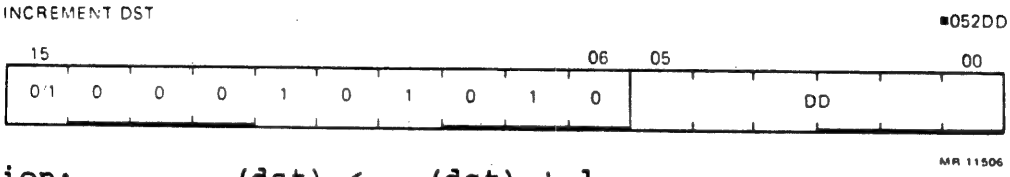
cleared otherwise
Z: set if result is 0; cleared otherwise
V: cleared
C: set

Description: Word: Replaces the contents of the destination address by their logical complement. (Each bit equal to 0 is set and each bit equal to 1 is cleared.)
Byte: Same.

Example:

```
COM R0

Before                  After

(R0) = 013333           (R0) = 164444

 NZVC                    NZVC
 0110                    1001
```

INC
INCB



Operation: (dst) <-- (dst) + 1

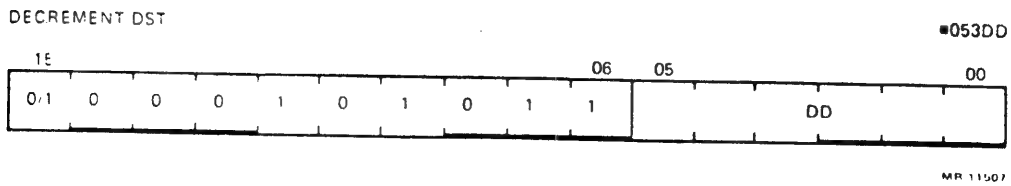
Condition Codes: N: set if result is < 0; cleared otherwise
Z: set if result is 0; cleared otherwise
V: set if (dst) held 077777; cleared otherwise
C: not affected

Description: Word: Add 1 to the contents of the destination.
Byte: Same.

Example:

```
INC R2

Before                  After

(R2) = 000333           (R2) = 000334

 NZVC                    NZVC
 0000                    0000
```

DEC
DECB



Operation: (dst) <-- (dst) - 1

Condition Codes: N: set if result is < 0; cleared otherwise

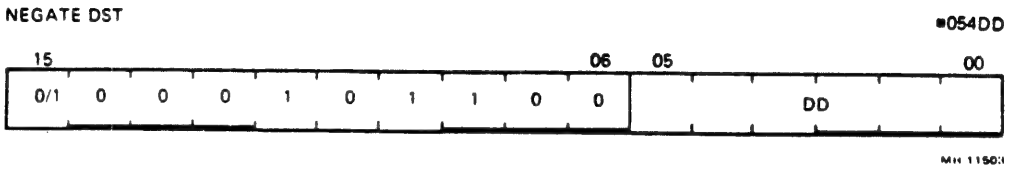
Z: set if result is 0; cleared otherwise
V: set if (dst) was 100000; cleared otherwise
C: not affected

Description: Word: Subtract 1 from the contents of the destination.
Byte: Same.

Example:

```
DEC R5

Before                  After

(R5) = 000001           (R5) = 000000

NZVC                    NZVC
1000                    0100
```

NEG
NEGB



Operation: (dst) <-- - (dst)

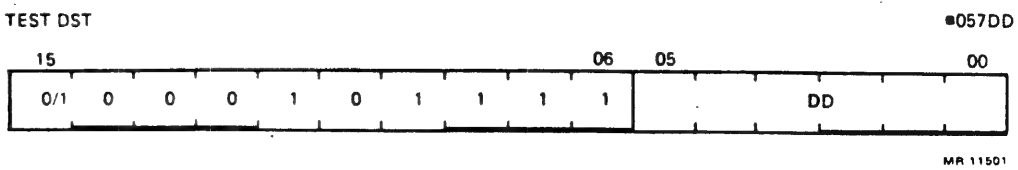
Condition Codes: N: set if result is < 0; cleared otherwise
Z: set if result is 0; cleared otherwise
V: set if result is 100000; cleared otherwise
C: cleared if result is 0; set otherwise

Description: Word: Replaces the contents of the destination address by its 2's complement. Note that 100000 is replaced by itself. (In 2's complement notation the most negative number has no positive counterpart.)
Byte: Same.

Example:

```
NEG R0

Before                  After

(R0) = 000010           (R0) = 177770

NZVC                    NZVC
0000                    1001
```

TST
TSTB



Operation: (dst) <-- (dst)

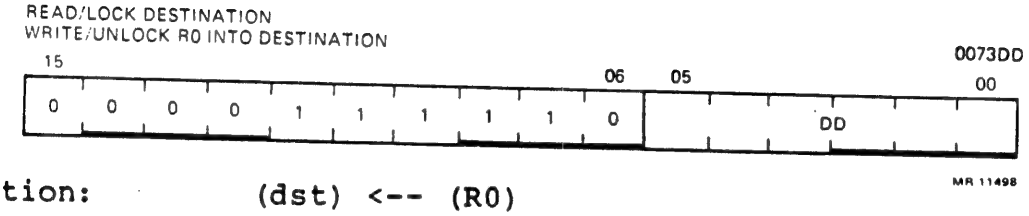
Condition Codes:
N: set if result is < 0; cleared otherwise
Z: set if result is 0; cleared otherwise
V: cleared
C: cleared

Description:
Word: Sets the condition codes N and Z according to the contents of the destination address; the contents of dst remain unmodified.
Byte: Same.

Example:

```
TST R1

Before              After

(R1) = 012340       (R1) = 012340

NZVC                NZVC
0011                0000
```

WRTLCK

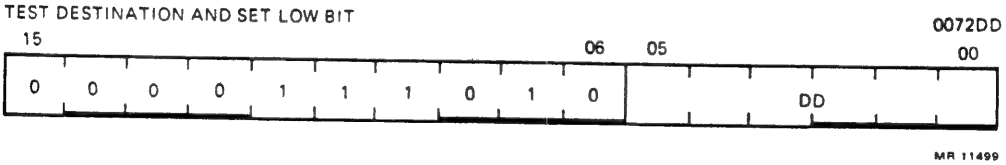


Operation: (dst) <-- (R0)

Condition Codes:
N: set if R0 < 0
Z: set if R0 = 0
V: cleared
C: unchanged

Description:
Writes contents of R0 into destination using bus lock. If mode is 0, traps to 10.

TSTSET



Operation: (R0) <-- (dst), (dst) <-- (dst) $\vee$ 000001 (octal)

Condition Codes:
N: set if R0 < 0
Z: set if R0 = 0
V: cleared
C: gets contents of destination bit 0.

Description:
Reads/locks destination word and stores it in R0. Writes/unlocks (R0) $\vee$ 1 into destination. If mode is 0, traps to 10.

6.3.4.2 Shifts And Rotates - Scaling data by factors of two is accomplished by the shift instructions:

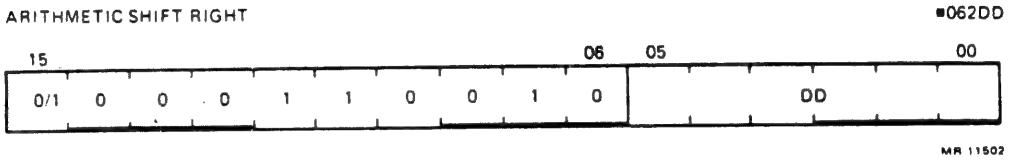
ASR -- Arithmetic shift right

ASL -- Arithmetic shift left

The sign bit (bit 15) of the operand is reproduced in shifts to the right. The low-order bit is filled with 0s in shifts to the left. Bits shifted out of the C bit, as shown in the following instructions, are lost.

The rotate instructions operate on the destination word and the C bit as though they formed a 17-bit "circular buffer." These instructions facilitate sequential bit testing and detailed bit manipulation.

ASR
ASRB

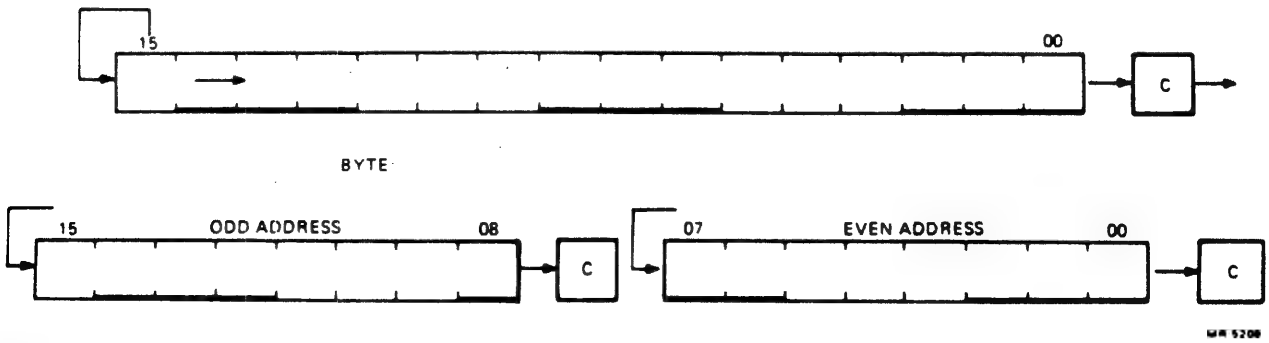


Operation: (dst) <-- (dst) shifted one place to the right

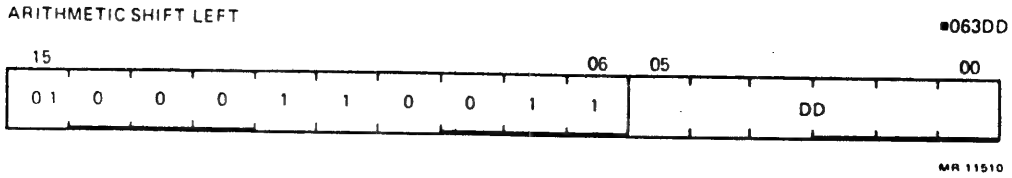
Condition Codes:
- N: set if high-order bit of result is set (result < 0); cleared otherwise
- Z: set if result = 0; cleared otherwise
- V: loaded from exclusive OR of N bit and C bit (as set by the completion of the shift operation)
- C: loaded from low-order bit of destination

Description:
Word: Shifts all bits of the destination right one place. Bit 15 is reproduced. The C bit is loaded from bit 0 of the destination. ASR performs signed division of the destination by 2.
Byte: Same.

Example:



ASL
ASLB

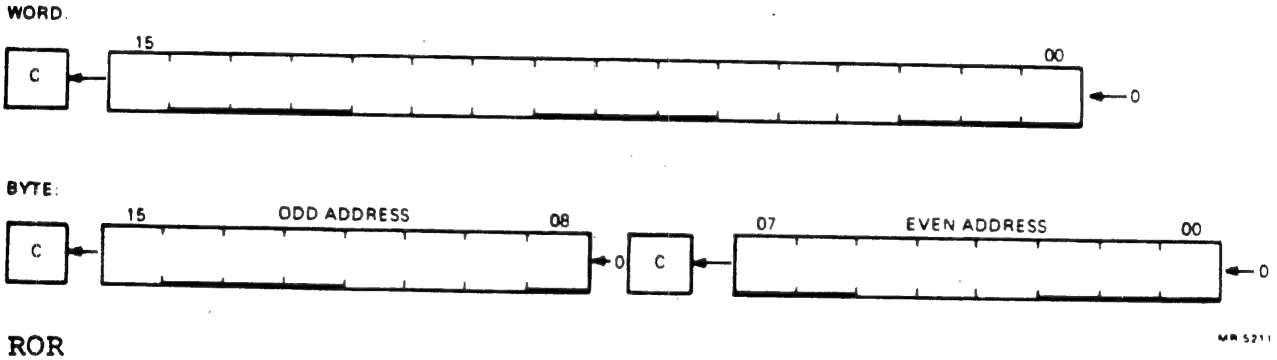


Operation: (dst) <-- (dst) shifted one place to the left

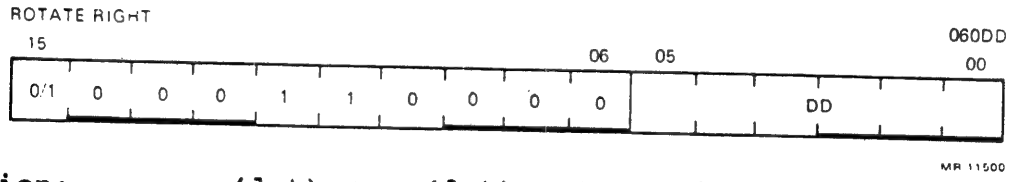
Condition Codes:
- N: set if high-order bit of result is set (result < 0); cleared otherwise
- Z: set if result = 0; cleared otherwise
- V: loaded with exclusive OR of N bit and C bit (as set by the completion of the shift operation)
- C: loaded with high-order bit of destination

Description: Word: Shifts all bits of the destination left one place. Bit 0 is loaded with a 0. The C bit of the status word is loaded from the most significant bit of the destination. ASL performs a signed multiplication of the destination by 2 with overflow indication. Byte: Same.

Example:



ROR
RORB



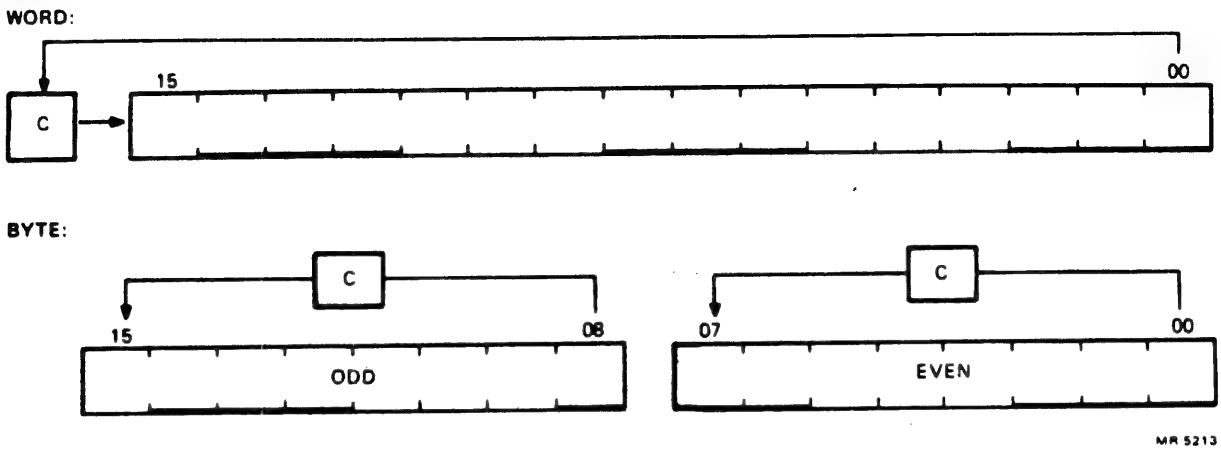
Operation: (dst) <-- (dst) rotate right one place

Condition Codes:
- N: set if high-order bit of result is set (result < 0); cleared otherwise
- Z: set if all bits of result = 0; cleared otherwise
- V: loaded with exclusive OR of N bit and C bit (as set by the completion of the rotate operation)
- C: loaded with low-order bit of destination

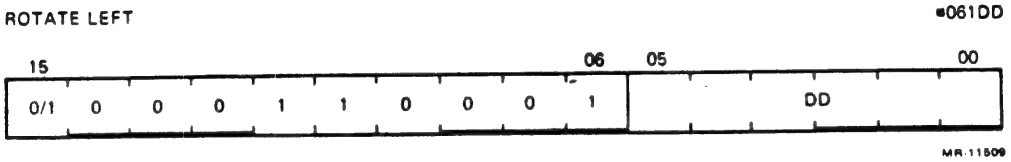
Description: Word: Rotates all bits of the destination right one place. Bit 0 is loaded into the C bit and the previous contents of the C bit are loaded

into bit 15 of the destination.
Byte: Same.

Example:



ROL
ROLB

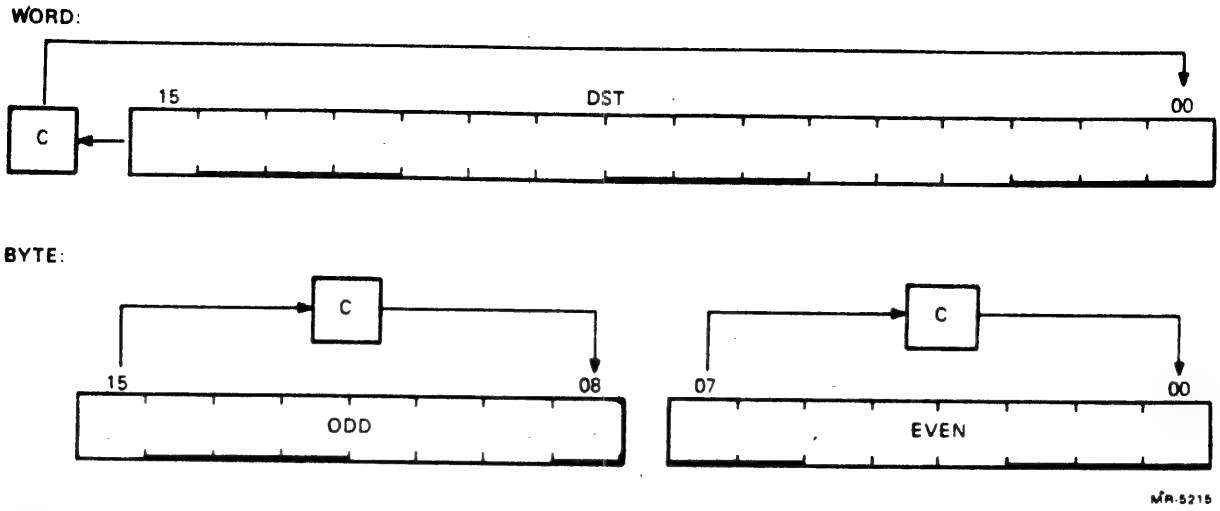


Operation: (dst) <-- (dst) rotate left one place

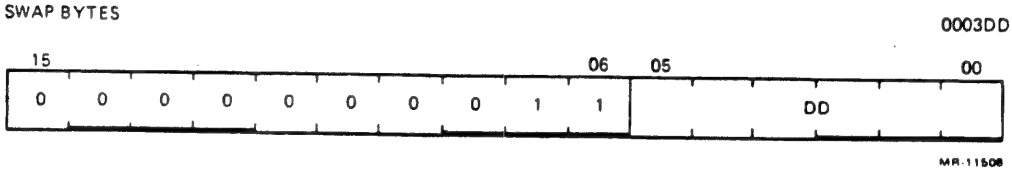
Condition Codes:
N: set if high-order bit of result word is set (result < 0); cleared otherwise
Z: set if all bits of result word = 0; cleared otherwise
V: loaded with exclusive OR of the N bit and C bit (as set by the completion of the rotate operation)
C: loaded with high-order bit of destination

Description: Word: Rotates all bits of the destination left one place. Bit 15 is loaded into the C bit of the status word and the previous contents of the C bit are loaded into bit 0 of the destination.
Byte: Same.

Example:



SWAB



Operation: byte 1/byte 0 <-- byte 0/byte 1

Condition Codes:
- N: set if high-order bit of low-order byte (bit 7) of result is set; cleared otherwise
- Z: set if low-order byte of result = 0; cleared otherwise
- V: cleared
- C: cleared

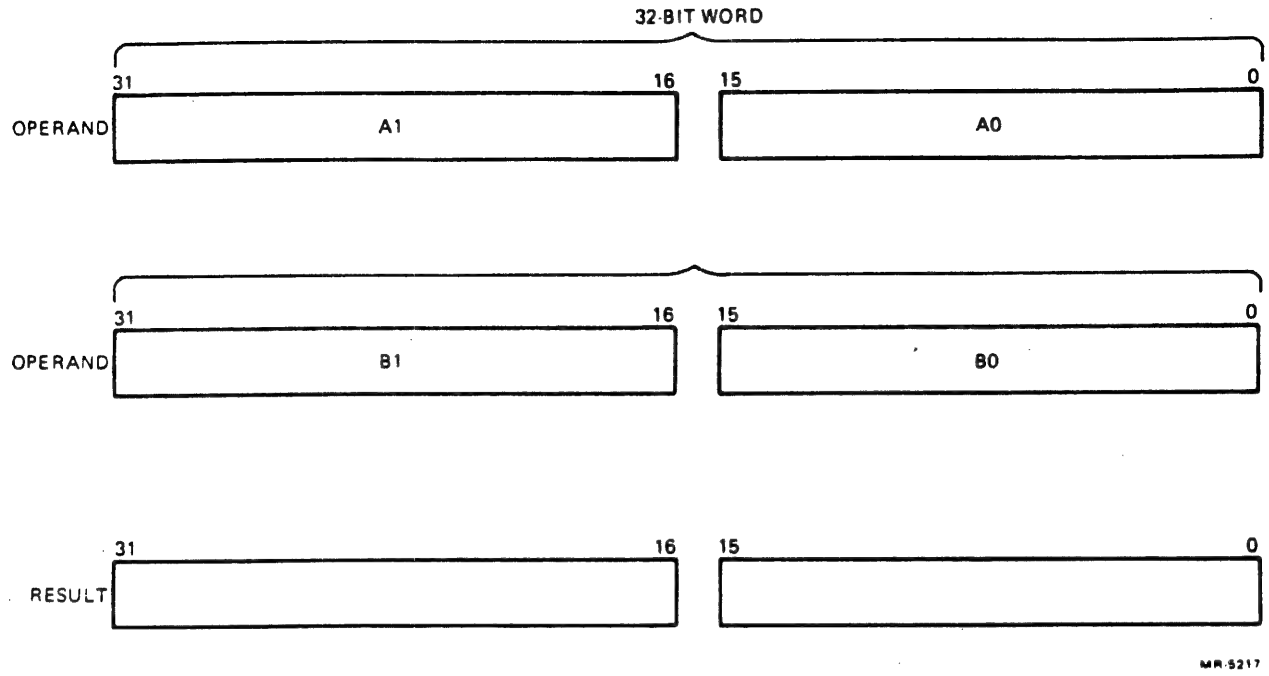
Description: Exchanges high-order byte and low-order byte of the destination word. (The destination must be a word address.)

Example:

```
SWAB R1

Before                  After

(R1) = 077777           (R1) = 177577

NZVC                    NZVC
1111                    0000
```

6.3.4.3 Multiple-Precision - It is sometimes necessary to do arithmetic operations on operands considered as multiple words or bytes. The DCJ11 makes special provision for such operations with the instructions ADC (add carry) and SBC (subtract carry) and their byte equivalents.

For example, two 16-bit words may be combined into a 32-bit double-precision word and added or subtracted as shown below.



Example:

The addition of -1 and -1 could be performed as follows.

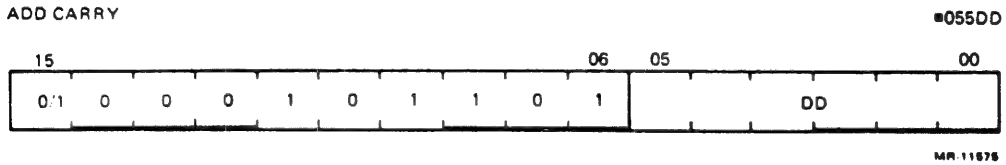
-1 = 37777777777

(R1) = 177777 (R2) = 177777 (R3) = 177777 (R4) = 177777

```
ADD R1,R2
ADC R3
ADD R4,R3
```

1. After (R1) and (R2) are added, 1 is loaded into the C bit.
2. The ADC instruction adds the C bit to (R3); (R3) = 0.
3. The (R3) and (R4) are added.
4. The result is 37777777776, or -2.

ADC
ADCB



Operation: (dst) <-- (dst) + (C bit)

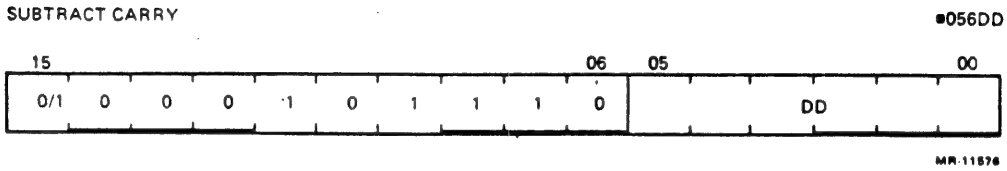
Condition Codes:
- N: set if result < 0; cleared otherwise
- Z: set if result = 0; cleared otherwise
- V: set if (dst) was 077777 and (C) was 1; cleared otherwise
- C: set if (dst) was 177777 and (C) was 1; cleared otherwise

Description: Word: Adds the contents of the C bit to the destination. This permits the carry from the addition of the low-order words to be carried to the high-order result.
Byte: Same.

Example: Double-precision addition may be done with the following instruction sequence.

```
ADD     A0,B0       ;add low-order parts
ADC     B1          ;add carry into high-order
ADD     A1,B1       ;add high-order parts
```

SBC
SBCB



Operation: (dst) <-- (dst) - (C)

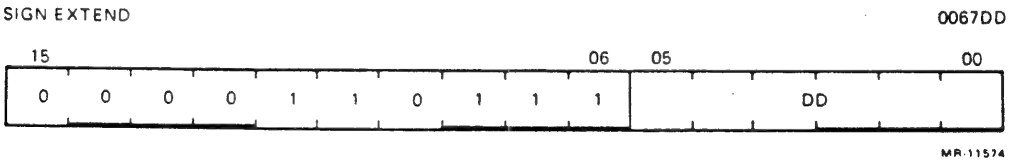
Condition Codes: N: set if result < 0; cleared otherwise
Z: set if result = 0; cleared otherwise
V: set if (dst) was 100000; cleared otherwise
C: set if (dst) was 0 and C was 1; cleared otherwise

Description: Word: Subtracts the contents of the C bit from the destination. This permits the carry from the subtraction of two low-order words to be subtracted from the high-order part of the result.
Byte: Same.

Example: Double-precision subtraction is done by:

```
SUB     A0,B0
SBC     B1
SUB     A1,B1
```

SXT



Operation: (dst) <-- 0 if N bit is clear
(dst) <-- 1 if N bit is set

Condition Codes: N: not affected
Z: set if N bit is clear
V: cleared
C: not affected

Description: If the condition code bit N is set, a -1 is

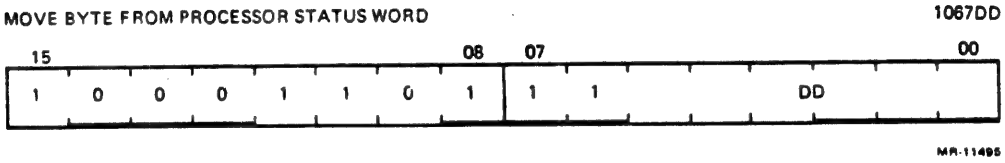
placed in the destination operand; if the N bit is clear, a 0 is placed in the destination operand. This instruction is particularly useful in multiple-precision arithmetic because it permits the sign to be extended through multiple words.

Example:

```
SXT A

Before                  After

(A) = 012345            (A) = 177777

NZVC                    NZVC
1000                    1000
```

6.3.4.4 PS Word Operators -

MFPS



Operation: (dst) <-- PS
dst lower 8 bits

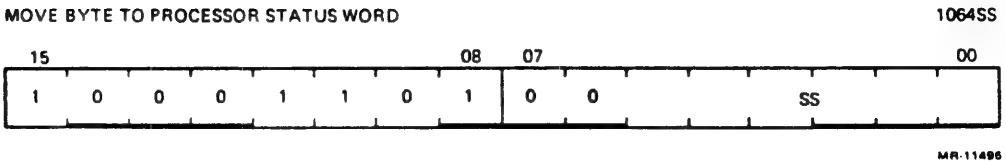
Condition Codes: N: set if PS <7> = 1; cleared otherwise
Z: set if PS <7:0> = 0; cleared otherwise
V: cleared
C: not affected

Description: The 8-bit contents of the PS are moved to the effective destination. If the destination is mode 0, PS bit 7 is sign-extended through the upper byte of the register. The destination operand address is treated as a byte address.

Example:

```
MFPS R0

Before                  After

R0 [0]                  R0 [000014]
PS [000014]             PS [000000]
```

MTPS



Operation: PS <-- (src)

Condition Codes: Set according to effective SRC operand bits <3:0>

Description: The eight bits of the effective operand replace the current contents of the lower byte of the PS. The source operand address is treated as a byte address. Note: The T bit (PS bit 4) cannot be set with this instruction. The SRC operand remains unchanged. This instruction can be used to change the priority bits (PS bits <7:5>) in the PS only in kernel mode. If not in kernel mode, PS bits <7:5> cannot be changed.

Example: MTPS R1

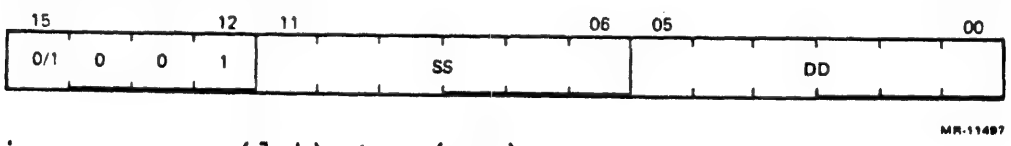
Before	After
(R1) = 000777	(R1) = 000777
(PS) = XXX000	(PS) = XXX357
NZVC	NZVC
0000	1111

6.3.5 Double-Operand Instructions - Double-operand instructions save instructions (and time) since they eliminate the need for "load" and "save" sequences such as those used in accumulator-oriented machines.

6.3.5.1 General -

MOV
MOVB

MOVE SOURCE TO DESTINATION



Operation: (dst) <-- (src)

Condition Codes:
N: set if (src) < 0; cleared otherwise
Z: set if (src) = 0; cleared otherwise
V: cleared
C: not affected

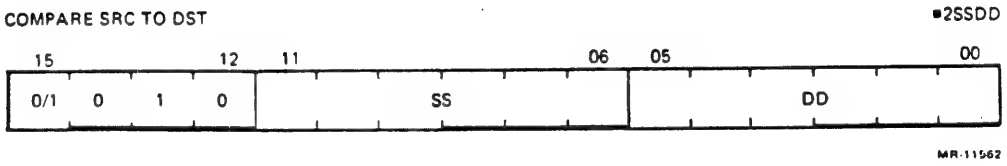
Description: Word: Moves the source operand to the destination location. The previous contents of the destination are lost. Contents of the source address are not affected.
Byte: Same as MOV. The MOVB to a register (unique among byte instructions) extends the most significant bit of the low-order byte (sign

extension). Otherwise, MOVB operates on bytes exactly as MOV operates on words.

Example:

```
MOV XXX,R1          ;loads register 1 with the contents of memory location; XXX represents a programmer-defined mnemonic used to represent a memory location

MOV #20,R0          ;loads the number 20 into register 0; # indicates that the value 20 is the operand

MOV @#20,-(R6)      ;pushes the operand contained in location 20 onto the stack

MOV (R6)+,@#177566  ;pops the operand off a stack and moves it into memory location 177566 (terminal print buffer)

MOV R1,R3           ;performs an inter-register transfer

MOVB @#177562,@#177566  ;moves a character from the terminal keyboard buffer to the terminal printer buffer
```

CMP
CMPB

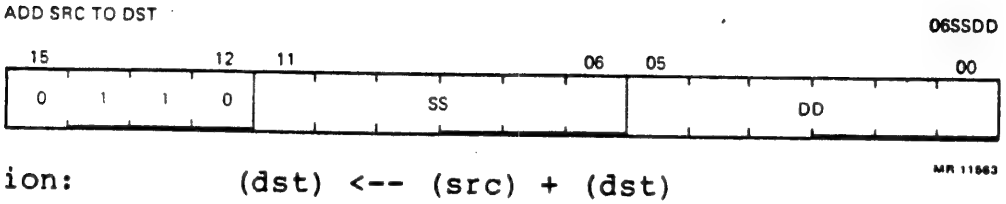


Operation: (src) - (dst)

Condition Codes:
- N: set if result < 0; cleared otherwise
- Z: set if result = 0; cleared otherwise
- V: set if there was arithmetic overflow; that is, operands were of opposite signs and the sign of the destination was the same as the sign of the result; cleared otherwise
- C: cleared if there was a carry from the result's most significant bit; set otherwise

Description: Compares the source and destination operands and sets the condition codes, which may then be used for arithmetic and logical conditional branches. Both operands are not affected. The only action is to set the condition codes. The compare is customarily followed by a conditional branch instruction. Note: Unlike the subtract instruction, the order of operation is (src) - (dst), not (dst) - (src).

ADD



Operation: (dst) <-- (src) + (dst)

Condition Codes:
N: set if result < 0; cleared otherwise
Z: set if result = 0; cleared otherwise
V: set if there was arithmetic overflow as a result of the operation; that is, both operands were of the same sign and the result was of the opposite sign; cleared otherwise
C: set if there was a carry from the result's most significant bit; cleared otherwise

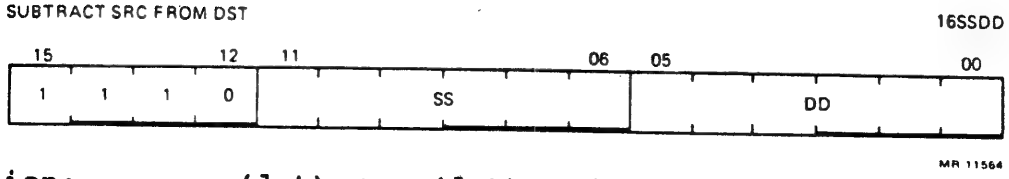
Description: Adds the source operand to the destination operand and stores the result at the destination address. The original contents of the destination are lost. The contents of the source are not affected. Two's complement addition is performed. Note: There is no equivalent byte mode.

Example:

Add to register:	ADD 20,R0
Add to memory:	ADD R1,XXX
Add register to register:	ADD R1,R2
Add memory to memory:	ADD @#17750,XXX

XXX is a programmer-defined mnemonic for a memory location.

SUB



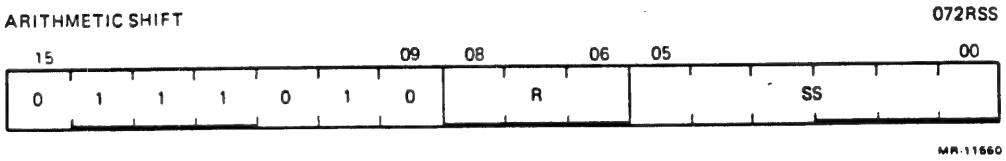
Operation: (dst) <-- (dst) - (src)

Condition Codes:
N: set if result < 0; cleared otherwise
Z: set if result = 0; cleared otherwise
V: set if there was arithmetic overflow as a result of the operation; that is, if operands were of opposite signs and the sign of the source was the same as the sign of the result; cleared otherwise
C: cleared if there was a carry from the result's most significant bit; set otherwise

Description:
Subtracts the source operand from the destination operand and leaves the result at the destination address. The original contents of the destination are lost. The contents of the source are not affected. In double-precision arithmetic the C bit, when set, indicates a "borrow." Note: There is no equivalent byte mode.

Example:
```
SUB R1,R2

Before                  After

(R1) = 011111           (R1) = 011111
(R2) = 012345           (R2) = 001234

 NZVC                    NZVC
 1111                    0000
```

ASH



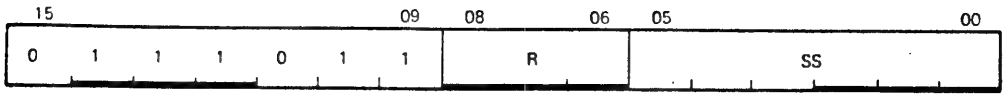
Operation:
R <-- R shifted arithmetically NN places to the right or left where NN = (src)

Condition Codes:
N: set if result < 0
Z: set if result = 0
V: set if sign of register changed during shift
C: loaded from last bit shifted out of register

Description:
The contents of the register are shifted right or left the number of times specified by the source operand. The shift count is taken as the low-order six bits of the source operand. This number ranges from -32 to +31. Negative is a right shift and positive is a left shift.

ASHC

ARITHMETIC SHIFT COMBINED ■073RSS



MR-11561

Operation: R, R ∨ 1 <-- R, R ∨ 1
The double word is shifted NN places to the right or left where NN = (src)

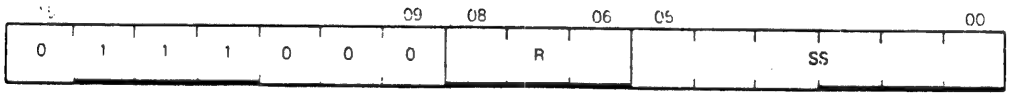
Condition Codes:
N: set if result < 0
Z: set if result = 0
V: set if sign bit changes during shift
C: loaded with high-order bit when left shift; loaded with low-order bit when right shift (loaded with the last bit shifted out of the 32-bit operand)

Description: The contents of the register and the register ORed with 1 are treated as one 32-bit word. R v 1 (bits<15:0>) and R (bits<31:16>) are shifted right or left the number of times specified by the shift count. The shift count is taken as the low-order six bits of the source operand. This number ranges from -32 to +31. Negative is a right shift and positive is a left shift.

When the register chosen is an odd number, the register and the register ORed with 1 are the same. In this case, the right shift becomes a rotate. The 16-bit word is rotated right the number of times specified by the shift count.

MUL

MULTIPLY 070RSS



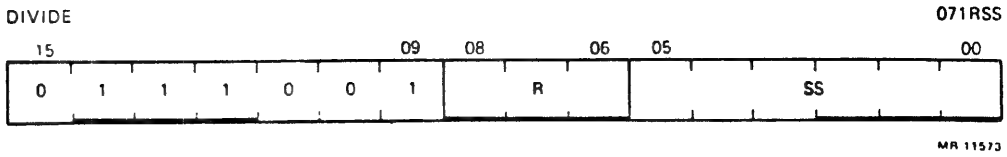
MR-11572

Operation: R, R ∨ 1 <-- R x (src)

Condition Codes:
N: set if product < 0
Z: set if product = 0
V: cleared
C: set if the result is less than -2 ** 15 or greater than or equal to 2 **15 - 1.

Description: The contents of the destination register and source taken as 2's complement integers are multiplied and stored in the destination register and the succeeding register, if R is even. If R is odd, only the low-order product is stored. Assembler syntax is: MUL S,R. (Note that the actual destination is R, R v 1, which reduces to just R when R is odd.

DIV



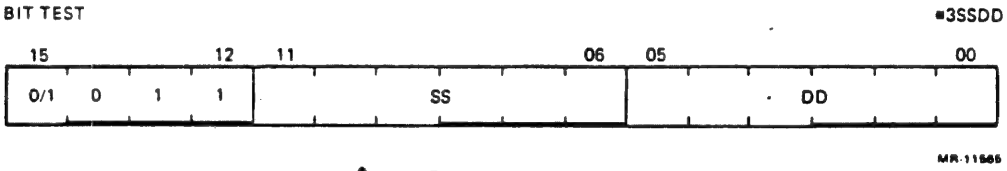
Operation: R, R ∨ 1 <-- R, R ∨ 1/(src)

Condition Codes:
N: set if quotient < 0
Z: set if quotient = 0
V: set if source = 0 or if the absolute value of the register is larger than the absolute value of the instruction in the source. (In this case the instruction is aborted because the quotient would exceed 15 bits.)
C: set if divide by zero is attempted.

Description: The 32-bit 2's complement integer in R and R v 1 is divided by the source operand. The quotient is left in R; the remainder is of the same sign as the dividend. R must be even.

6.3.5.2 Logical - These instructions have the same format a those in the double-operand arithmetic group. They permi operations on data at the bit level.

BIT
BITB



Operation: (src) ∧ (dst)

Condition Codes:
N: set if high-order bit of result set; cleared otherwise
Z: set if result = 0; cleared otherwise
V: cleared
C: not affected

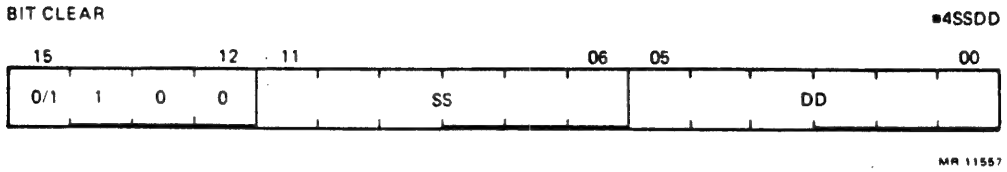
Description: Performs logical AND comparison of the source and destination operands and modifies condition codes accordingly. Neither the source nor the destination is affected. The BIT instruction may be used to test whether any of the corresponding bits set in the destination are also set in the source, or whether all corresponding bits set in the destination are clear in the source.

Example: BIT #30,R3 ;test bits three and four of R3 to see if both are off.

R3 = 0 000 000 000 011 000

Before	After
NZVC	NZVC
1111	0001

BIC
BICB



Operation: (dst) <-- ~(src) ∧ (dst)

Condition Codes:
N: set if high-order bit of result set; cleared otherwise
Z: set if result = 0; cleared otherwise
V: cleared
C: not affected

Description: Clears each bit in the destination that corresponds to a set bit in the source. The original contents of the destination are lost. The contents of the source are not affected.

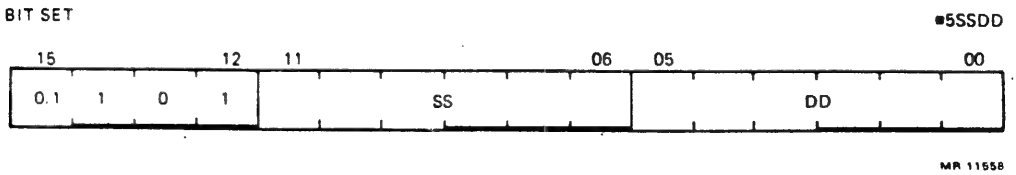
Example: BIC R3,R4

Before	After
(R3) = 001234	(R3) = 001234
(R4) = 001111	(R4) = 000101
NZVC	NZVC
1111	0001

Before: (R3) = 0 000 001 010 011 100
(R4) = 0 000 001 001 001 001

After: (R4) = 0 000 000 001 000 001

BIS
BISB



Operation: (dst) <-- (src) ∨ (dst)

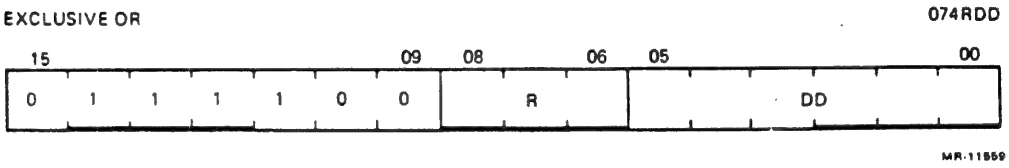
Condition Codes:
N: set if high-order bit of result set; cleared otherwise
Z: set if result = 0; cleared otherwise
V: cleared
C: not affected

Description: Performs an inclusive OR operation between the source and destination operands and leaves the result at the destination address; that is, corresponding bits set in the source are set in the destination. The contents of the destination are lost.

Example:

```
BIS R0,R1

Before                  After

(R0) = 001234           (R0) = 001234
(R1) = 001111           (R1) = 001335

NZVC                    NZVC
0000                    0000

Before:  (R0) = 0 000 001 010 011 100
         (R1) = 0 000 001 001 001 001

After:   (R1) = 0 000 001 011 011 101
```

XOR



Operation: (dst) <-- (reg) $\forall$ (dst)

Condition Codes:
N: set if result < 0; cleared otherwise
Z: set if result = 0; cleared otherwise
V: cleared
C: not affected

Description: The exclusive OR of the register and destination operand is stored in the destination address. The contents of the register are not affected. The assembler format is XOR R,D.

Example:

```
XOR R0,R2

Before                  After

(R0) = 001234           (R0) = 001234
(R2) = 001111           (R2) = 000325

NZVC                    NZVC
1111                    0001
```

```
Before:   (R0) = 0 000 001 010 011 100
          (R2) = 0 000 001 001 001 001

After:    (R2) = 0 000 000 011 010 101
```

6.3.6 Program Control Instructions - The following paragraphs describe the DCJ11 instructions that affect program control.

6.3.6.1 Branches - These instructions cause a branch to a location defined by the sum of the offset (multiplied by 2) and the current contents of the program counter if:

1. The branch instruction is unconditional.

2. It is conditional and the conditions are met after testing the condition codes (NZVC).

The offset is the number of words from the current contents of the PC, forward or backward. Note that the current contents of the PC point to the word following the branch instruction.

Although the offset expresses a byte address, the PC is expressed in words. The offset is automatically multiplied by 2 and sign-extended to express words before it is added to the PC. Bit 7 is the sign of the offset. If it is set, the offset is negative and the branch is done in the backward direction. If it is not set, the offset is positive and the branch is done in the forward direction.

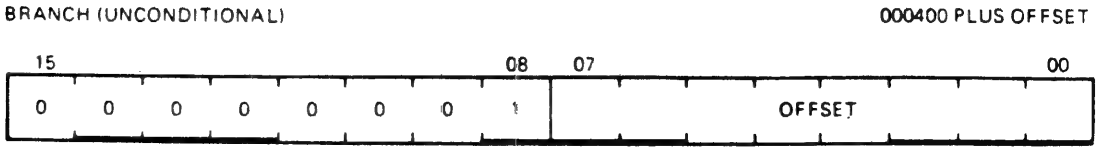
The 8-bit offset allows branching in the backward direction by 200 (octal) words (400 octal bytes) from the current PC, and in the forward direction by 177 (octal) words (376 octal bytes) from the current PC.

The DCJ11 assembler typically handles address arithmetic for the user and computes and assembles the proper offset field for branch instructions in the form:

```
Bxx loc
```

Bxx is the branch instruction and loc is the address to which the branch is to be made. The assembler gives an error indication in the instruction if the permissible branch range is exceeded. Branch instructions have no effect on condition codes. Conditional branch instructions where the branch condition is not met are treated as NOPs.

BR



Operation: PC <-- PC + (2 × offset)

Condition Codes: Not affected

Description: Provides a way of transferring program control within a range of -128 to +127 words with a one word instruction.

New PC address = updated PC + (2 × offset)

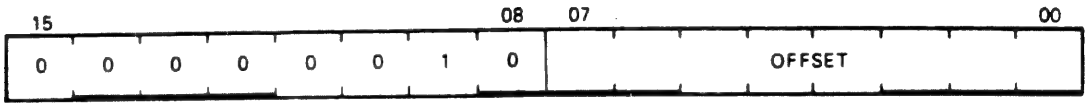
Updated PC = address of branch instruction +2

Example: With the branch instruction at location 500, the following offsets apply.

New PC Address	Offset Code	Offset (decimal)
474	375	-3
476	376	-2
500	377	-1
502	000	0
504	001	+1
506	002	+2

BNE

BRANCH IF NOT EQUAL (TO ZERO) 001000 PLUS OFFSET



Operation: PC <-- PC + (2 × offset) if Z = 0

Condition Codes: Not affected

Description: Tests the state of the Z bit and causes a branch if the Z bit is clear. BNE is the complementary operation of BEQ. It is used to test: (1) inequality following a CMP, (2) that some bits set in the destination were also in the source following a BIT operation, and (3) generally, that the result of the previous operation was not 0.

Example: Branch to C if A ≠ B

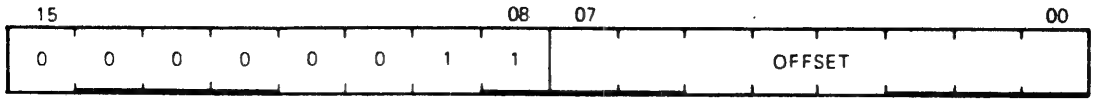
```
CMP A,B          ;compare A and B
BNE C            ;branch if they are not equal
```

Branch to C if A + B ≠ 0

```
ADD A,B          ;add A to B
BNE C            ;branch if the result is not
                     equal to 0
```

BEQ

BRANCH IF EQUAL (TO ZERO) 001400 PLUS OFFSET



Operation: PC <-- PC + (2 × offset) if Z = 1

Condition Codes: Not affected

Description: Tests the state of the Z bit and causes a branch if Z is set. It is used to test: (1) equality following a CMP operation, (2) that no bits set in the destination were also set in the source following a BIT operation, and (3) generally, that the result of the previous operation was 0.

Example: Branch to C if A = B (A - B = 0)

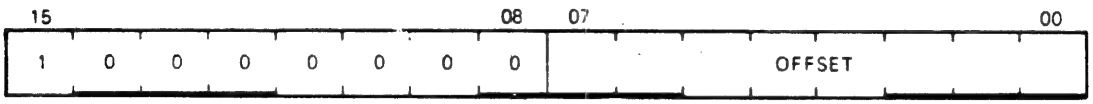
```
CMP A,B                 ;compare A and B
BEQ C                   ;branch if they are equal
```

Branch to C if A + B = 0

```
ADD  A,B                ;add A to B
BEQ  C                  ;branch if the result = 0
```

BPL

BRANCH IF PLUS 100000 PLUS OFFSET



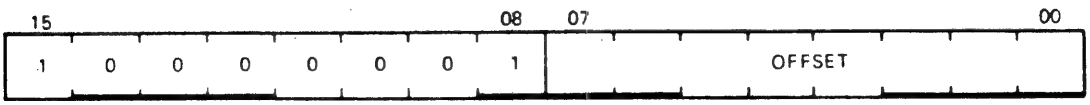
Operation: PC <-- PC + (2 × offset) if N = 0

Condition Codes: Not affected

Description: Tests the state of the N bit and causes a branch if N is clear (positive result). BPL is the complementary operation of BMI.

BMI

BRANCH IF MINUS 100400 PLUS OFFSET



Operation: PC <-- PC + (2 × offset) if N = 1

Condition Codes: Not affected

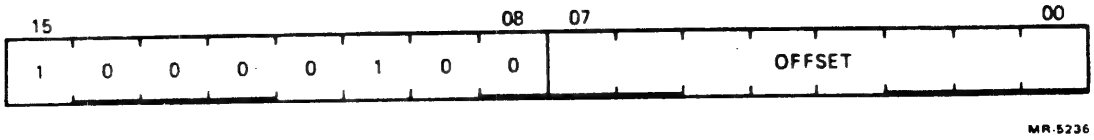
Description: Tests the state of the N bit and causes a branch if N is set. It is used to test the sign (most

significant bit) of the result of the previous operation), branching if negative. BMI is the complementary function of BPL.

BVC

BRANCH IF OVERFLOW IS CLEAR

102000 PLUS OFFSET



Operation: PC <-- PC + (2 × offset) if V = 0

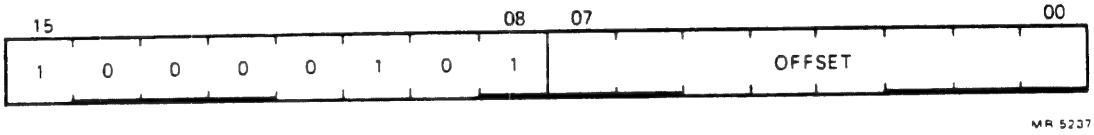
Condition Codes: Not affected

Description: Tests the state of the V bit and causes a branch if the V bit is clear. BVC is complementary operation to BVS.

BVS

BRANCH IF OVERFLOW IS SET

102400 PLUS OFFSET



Operation: PC <-- PC + (2 × offset) if V = 1

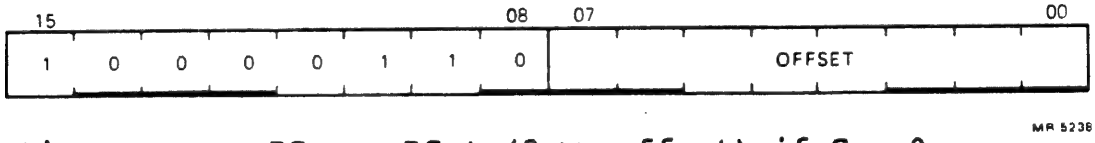
Condition Codes: Not affected

Description: Tests the state of the V bit (overflow) and causes a branch if V is set. BVS is used to detect arithmetic overflow in the previous operation.

BCC

BRANCH IF CARRY IS CLEAR

103000 PLUS OFFSET



Operation: PC <-- PC + (2 × offset) if C = 0

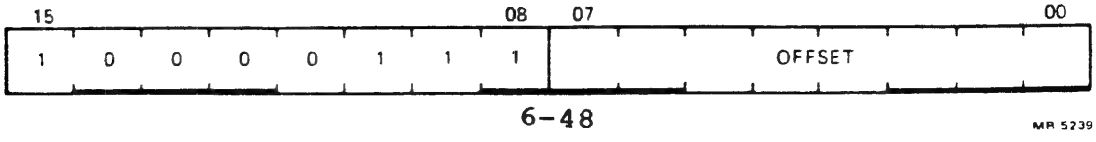
Condition Codes: Not affected

Description: Tests the state of the C bit and causes a branch if C is clear. BCC is the complementary operation of BCS.

BCS

BRANCH IF CARRY IS SET

103400 PLUS OFFSET



Operation: PC <-- PC + (2 × offset) if C = 1

Condition Codes: Not affected

Description: Tests the state of the C bit and causes a branch if C is set. It is used to test for a carry in the result of a previous operation.

6.3.6.2 Signed Conditional Branches - Particular combinations of the condition code bits are tested with the signed conditional branches. These instructions are used to test the results of instructions in which the operands were considered as signed (2's complement) values.

Note that the sense of signed comparisons differs from that of unsigned comparisons in that in signed, 16-bit, 2's complement arithmetic the sequence of values is as follows.

largest positive	077777
	077776
	.
	.
	.
	000001
	000000
	177777
	177776
	.
	.
	.
smallest negative	100001
	100000

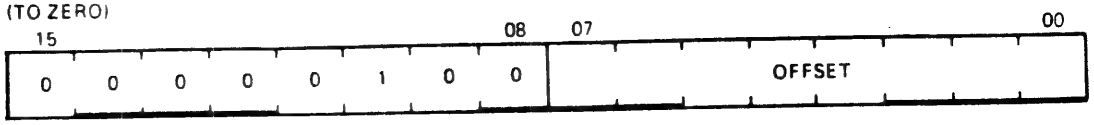
Whereas, in unsigned, 16-bit arithmetic, the sequence is considered to be:

highest	177777
	.
	.
	.
	.
	.
	.
	000002
	000001
lowest	000000

BGE

BRANCH IF GREATER THAN OR EQUAL (TO ZERO)

002000 PLUS OFFSET



Operation: PC <-- PC + (2 × offset) if N ⊻ V = 0

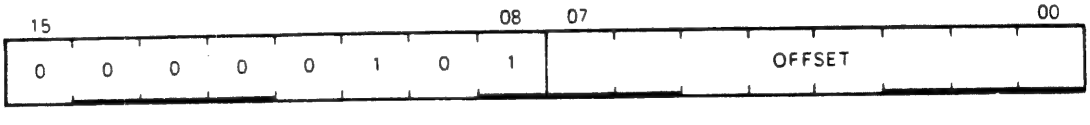
Condition Codes: Not affected

Description: Causes a branch if N and V are either both clear or both set. BGE is the complementary operation of BLT. Thus, BGE will always cause a branch when it follows an operation that caused addition of two positive numbers. BGE will also cause a branch on a 0 result.

BLT

BRANCH IF LESS THAN (ZERO)

002400 PLUS OFFSET



Operation: PC <-- PC + (2 × offset) if N ⊻ V = 1

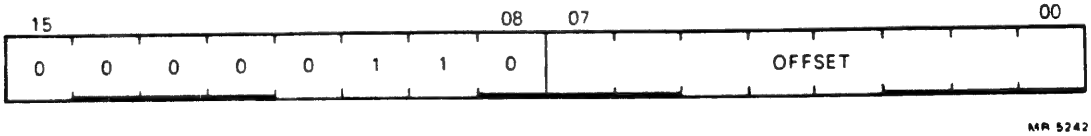
Condition Codes: Not affected

Description: Causes a branch if the exclusive OR of the N and V bits is one. Thus, BLT will always branch following an operation that added two negative numbers, even if overflow occurred. In particular, BLT will always cause a branch if it follows a CMP instruction operating on a negative source and a positive destination (even if overflow occurred). Further, BLT will never cause a branch when it follows a CMP instruction operating on a positive source and negative destination. BLT will not cause a branch if the result of the previous operation was 0 (without overflow).

BGT

BRANCH IF GREATER THAN (ZERO)

003000 PLUS OFFSET



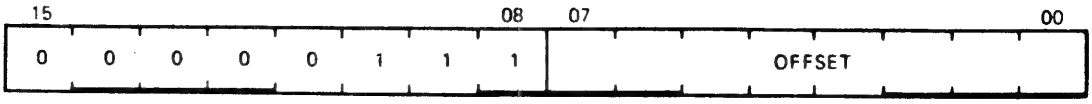
Operation: PC <-- PC + (2 × offset) if Z ∨ (N ⊻ V) = 0

Condition Codes: Not affected

Description: Operation of BGT is similar to BGE, except that BGT will not cause a branch on a 0 result.

BLE

BRANCH IF LESS THAN OR EQUAL (TO ZERO) 003400 PLUS OFFSET



Operation: PC <-- PC + (2 × offset) if Z $\vee$ (N $\veebar$ V) = 1 MR 5243

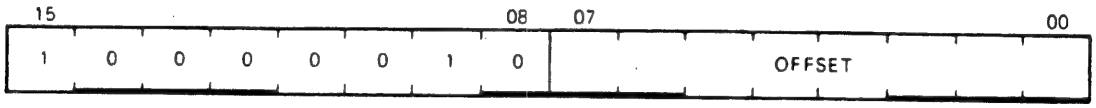
Condition Codes: Not affected

Description: Operation is similar to BLT, but in addition will cause a branch if the result of the previous operation was 0.

6.3.6.3 Unsigned Conditional Branches - The unsigned conditional branches provide a means for testing the result of comparison operations in which the operands are considered as unsigned values.

BHI

BRANCH IF HIGHER 101000 PLUS OFFSET



MR 5244

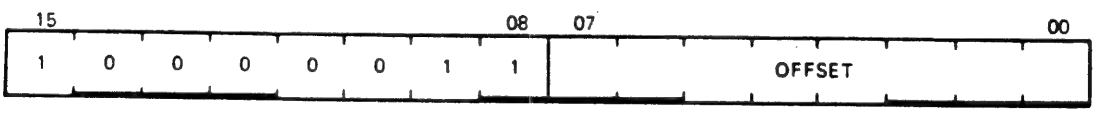
Operation: PC <-- PC + (2 × offset) if C = 0 and Z = 0

Condition Codes: Not affected

Description: Causes a branch if the previous operation caused neither a carry nor a 0 result. This will happen in comparison (CMP) operations as long as the source has a higher unsigned value than the destination.

BLOS

BRANCH IF LOWER OR SAME 101400 PLUS OFFSET



MR-5245

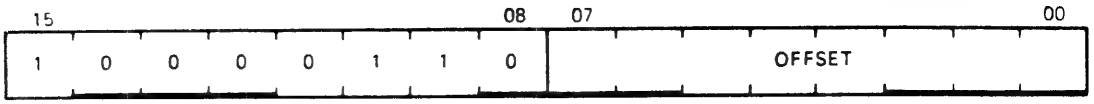
Operation: PC <-- PC + (2 × offset) if C $\vee$ Z = 1

Condition Codes: Not affected

Description: Causes a branch if the previous operation caused either a carry or a 0 result. BLOS is the complementary operation of BHI. The branch will occur in comparison operations as long as the source is equal to or has a lower unsigned value than the destination.

BHIS

BRANCH IF HIGHER OR SAME 103000 PLUS OFFSET



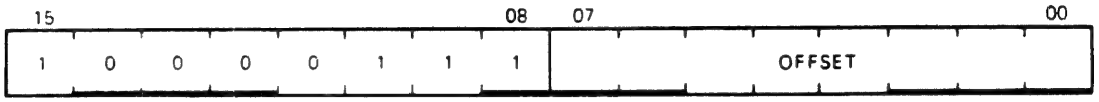
Operation: PC <-- PC + (2 × offset) if C = 0

Condition Codes: Not affected

Description: BHIS is the same instruction as BCC. This mnemonic is included for convenience only.

BLO

BRANCH IF LOWER 103400 PLUS OFFSET



Operation: PC <-- PC + (2 × offset) if C = 1

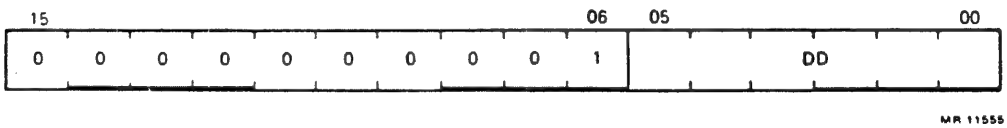
Condition Codes: Not affected

Description: BLO is the same instruction as BCS. This mnemonic is included for convenience only.

6.3.6.4 Jump And Subroutine Instructions - The subroutine call in the DCJ11 provides for automatic nesting of subroutines, reentrancy, and multiple entry points. Subroutines may call other subroutines (or indeed themselves) to any level of nesting without making special provision for storage of return addresses at each level of subroutine call. The subroutine calling mechanism does not modify any fixed location in memory, and thus provides for reentrancy. This allows one copy of a subroutine to be shared among several interrupting processes.

JMP

JUMP 0001DD



Operation: PC <-- (dst)

Condition Codes: Not affected

Description: JMP provides more flexible program branching than the branch instructions do. Control may be transferred to any location in memory (no range limitation) and can be accomplished with the full flexibility of the addressing modes, with the exception of register mode 0. Execution of

a jump with mode 0 will cause an "illegal instruction" condition, and will cause the CPU to trap to vector address four. (Program control cannot be transferred to a register.) Register-deferred mode is legal and will cause program control to be transferred to the address held in the specified register. Note that instructions are word data and must therefore be fetched from an even-numbered address.

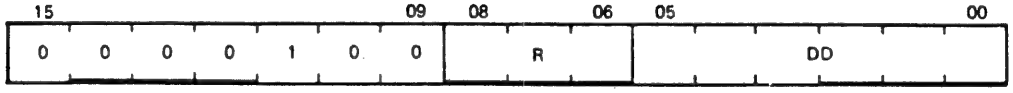
Deferred-index mode JMP instructions permit transfer of control to the address contained in a selectable element of a table of dispatch vectors.

Example:

```
First:

JMP FIRST              ;transfers to FIRST
. . . . .
. . . . .
JMP @LIST              ;transfers to location
                        pointed  to at LIST
. . . . .

List:

FIRST                  ;pointer to FIRST

JMP @(SP)+             ;transfer to location
                        pointed to by the top of
                        the stack, and remove the
                        pointer from the stack
```

JSR

JUMP TO SUBROUTINE 004RDD



MR-11556

Operation:

(tmp) <-- (dst) (tmp is an internal processor register)

↓ (SP) <-- reg (Push reg contents onto processor stack)

reg <-- PC (PC holds location following JSR; this address now put in reg)

PC <-- (dst) (PC now points to subroutine destination)

Description:

In execution of the JSR, the old contents of the specified register (the "linkage pointer") are automatically pushed onto the processor stack and new linkage information is placed in the register. Thus, subroutines nested within subroutines to any depth may all be called with

the same linkage register. There is no need either to plan the maximum depth at which any particular subroutine will be called or to include instructions in each routine to save and restore the linkage pointer. Further, since all linkages are saved in a reentrant manner on the processor stack, execution of a subroutine may be interrupted. The same subroutine may be reentered and executed by an interrupt service routine. Execution of the initial subroutine can then be resumed when other requests are satisfied. This process (called "nesting") can proceed to any level.

A subroutine called with a JSR reg,dst instruction can access the arguments following the call with either autoincrement addressing, (reg) +, if arguments are accessed sequentially, or by indexed addressing, X(reg), if accessed in random order. These addressing modes may also be deferred, @(reg)+ and @X(reg), if the parameters are operand addresses rather than the operands themselves.

JSR PC, dst is a special case of the DCJ11 subroutine call suitable for subroutine calls that transmit parameters through the general registers. The SP and the PC are the only registers that may be modified by this call.

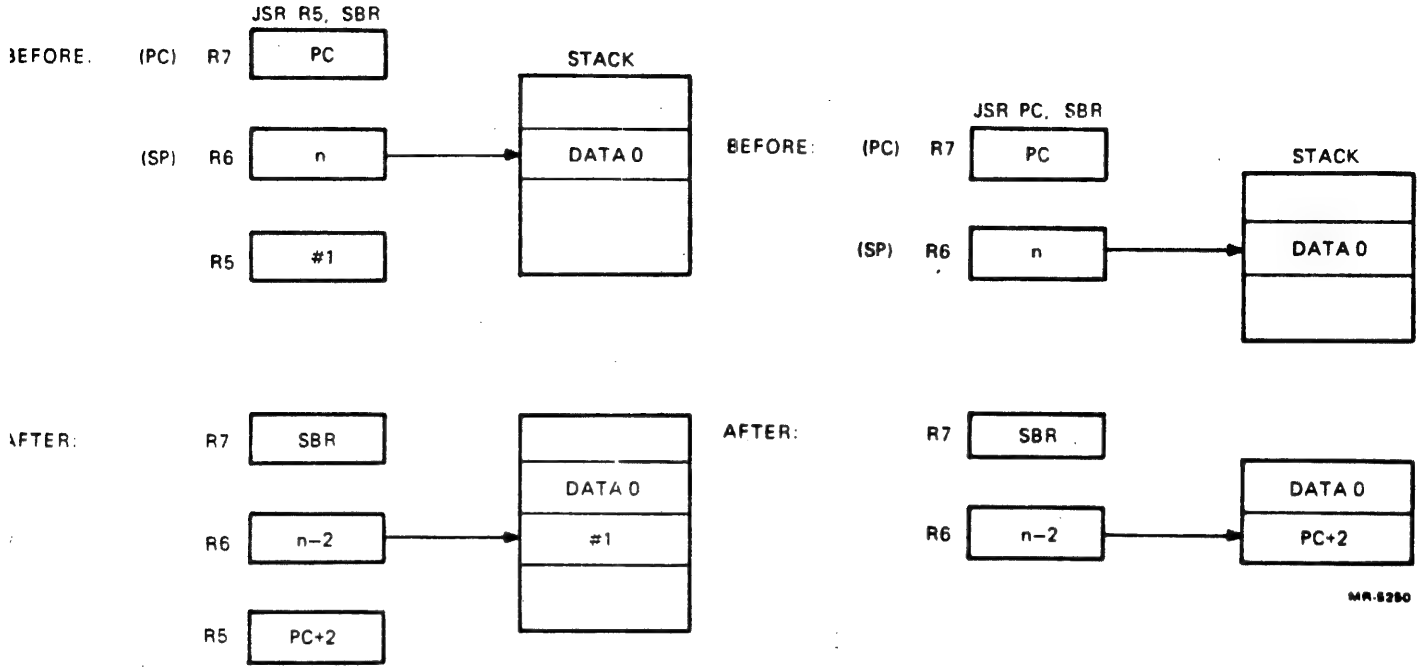
Another special case of the JSR instruction is JSR PC,@(SP) +, which exchanges the top element of the processor stack with the contents of the program counter. This instruction allows two routines to swap program control and resume operation from where they left off when they are recalled. Such routines are called "coroutines."

Return from a subroutine is done by the RTS instruction. RTS reg loads the contents of reg into the PC and pops the top element of the processor stack into the specified register.

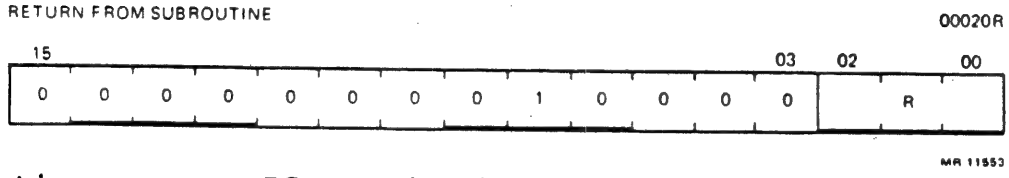
Example:

```
                                        R5              R6      R7
SBCALL:         JSR R5,SBR              #1              n       SBCALL
SBCALL+4:       ARG 1
                ARG 2
                .
                .
SBCALL+2+2M:    ARG M
CONT:           Next Instruction        #1              n       CONT
                .
                .
SBR:            MOV (R5)+,dst 1         SBCALL+4        n-2     SBR
                MOV (R5)+,dst 2
                .
```

```
                MOV (R5)+,dst M     SBCALL+2+2M
                Other Instructions  CONT
EXIT:           RTS R5              CONT                n-2    EXIT
```



RTS

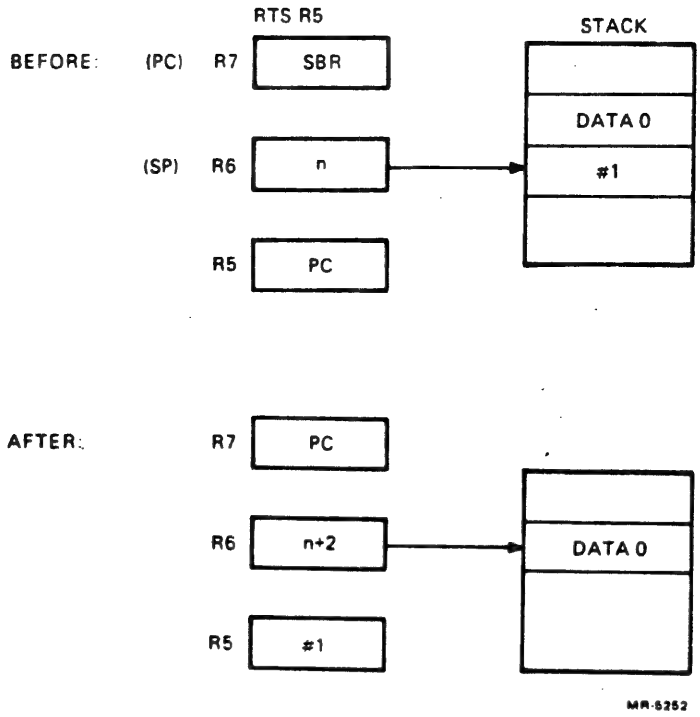


Operation: PC <-- (reg)
(reg) <-- (SP) ↑

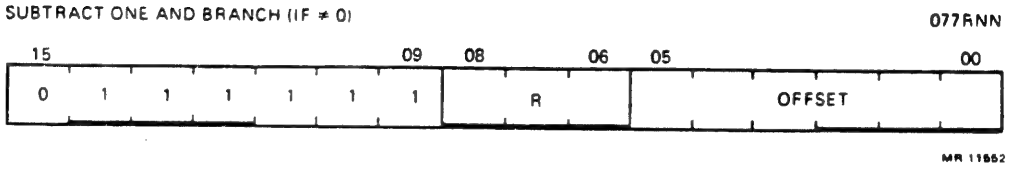
Description: Loads the contents of the register into PC and pops the top element of the processor stack into the specified register.

Return from a nonreentrant subroutine is typically made through the same register that was used in its call. Thus, a subroutine called with a JSR PC, dst exits with a RTS PC and a subroutine called with a JSR R5, dst, may pick up parameters with addressing modes (R5) +, X(R5), or @X(R5) and finally exits, with an RTS R5.

Example: RTS R5



SOB



Operation: (R) <-- (R) - 1; if this result ≠ 0, then PC <-- PC - (2 x offset); if (R) = 0 then PC <-- PC

Condition Codes: Not affected

Description: The register is decremented. If the contents does not equal 0, twice the offset is subtracted from the PC (now pointing to the following word). The offset is interpreted as a 6-bit positive number. This instruction provides a fast, efficient method of loop control. The assembler syntax is SOB R,A where A is the address to which transfer is to be made if the decremented R is not equal to 0. Note: the SOB instruction cannot be used to transfer control in the forward direction.

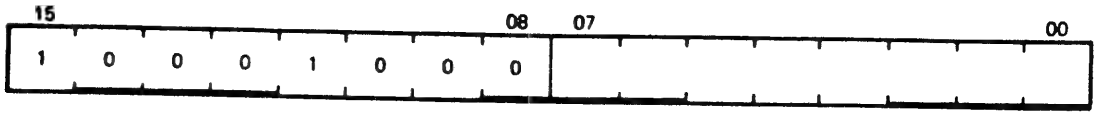
6.3.6.5 Traps - Trap instructions provide for calls to emulators, I/O monitors, debugging packages, and user-defined interpreters. A trap is effectively an interrupt generated by software. When a trap occurs, the contents of the current program counter (PC) and processor status word (PS) are pushed onto the processor stack and replaced by the contents of a 2-word trap vector containing a new PC and new PS. The return sequence from a trap involves executing an RTI or RTT instruction, which restores the old PC and old PS by popping them from the stack. Trap instruction vectors are located

at permanently assigned fixed addresses.

EMT

EMULATOR TRAP

104000–104377



Operation:

```
↓ (SP) <-- PS
↓ (SP) <-- PC
PC <-- (30)
PS <-- (32)
```

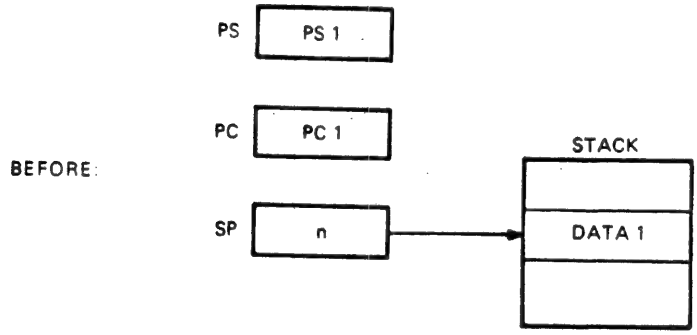
Condition Codes:

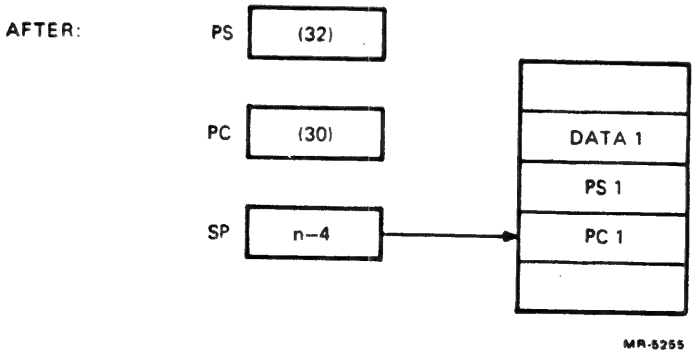
N: loaded from trap vector
Z: loaded from trap vector
V: loaded from trap vector
C: loaded from trap vector

Description:

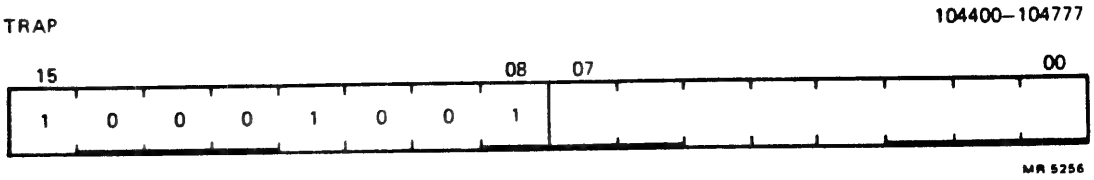
All operation codes from 104000 to 104377 are EMT instructions and may be used to transmit information to the emulating routine (e.g., function to be performed). The trap vector for EMT is at address 30. The new PC is taken from the word at address 30; the new processor status (PS) is taken from the word at address 32.

CAUTION: EMT is used frequently by DIGITAL system software and is therefore not recommended for general use.





TRAP



Operation:

```
↓ (SP) <-- PS
↓ (SP) <-- PC
PC <-- (34)
PS <-- (36)
```

Condition Codes:

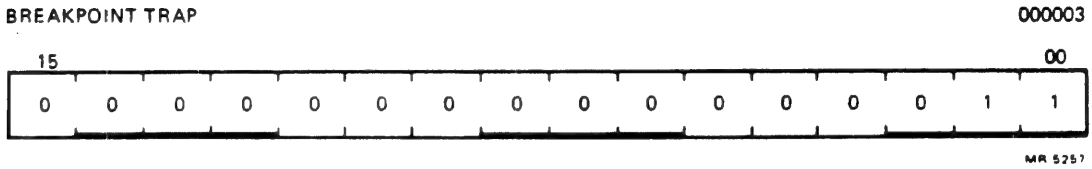
N: loaded from trap vector
Z: loaded from trap vector
V: loaded from trap vector
C: loaded from trap vector

Description:

Operation codes from 104400 to 104777 are TRAP instructions. TRAPs and EMTs are identical in operation, except that the trap vector for TRAP is at address 34.

NOTE: Since DIGITAL software makes frequent use of EMT, the TRAP instruction is recommended for general use.

BPT



Operation:

```
↓ (SP) <-- PS
↓ (SP) <-- PC
PC <-- (14)
PS <-- (16)
```

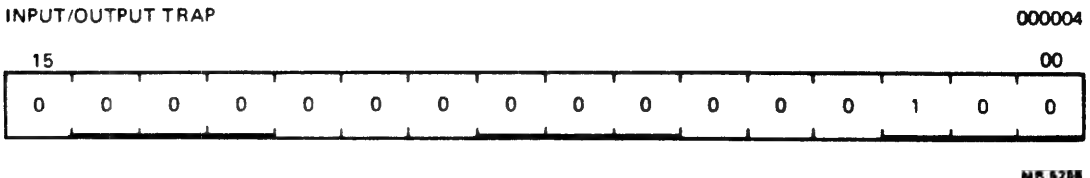
Condition Codes:

N: loaded from trap vector
Z: loaded from trap vector
V: loaded from trap vector
C: loaded from trap vector

Description:

Performs a trap sequence with a trap vector address of 14. Used to call debugging aids. The user is cautioned against employing code 000003 in programs run under these debugging aids. (No information is transmitted in the low byte.)

IOT



Operation:

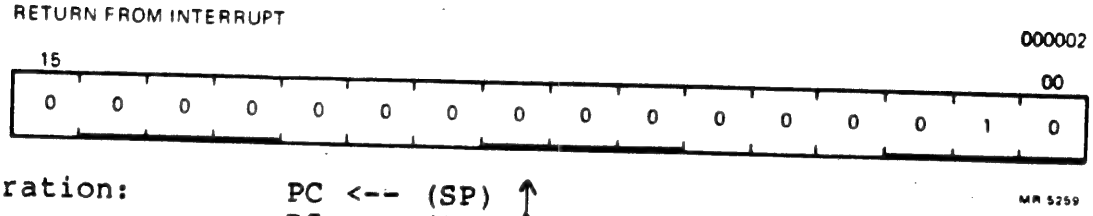
```
↓ (SP) <-- PS
↓ (SP) <-- PC
```

PC <-- (20)
PS <-- (22)

Condition Codes:
N: loaded from trap vector
Z: loaded from trap vector
V: loaded from trap vector
C: loaded from trap vector

Description: Performs a trap sequence with a trap vector address of 20. (No information is transmitted in the low byte.)

RTI

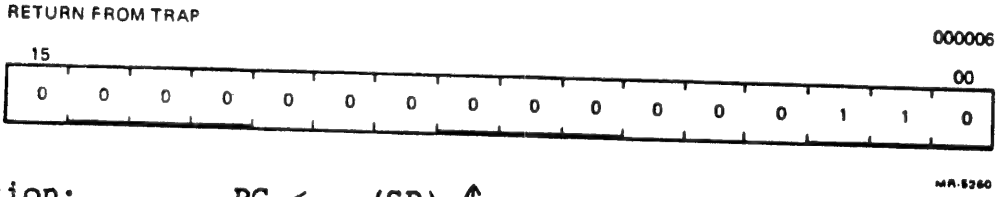


Operation:
PC <-- (SP) ↑
PS <-- (SP) ↑

Condition Codes:
N: loaded from processor stack
Z: loaded from processor stack
V: loaded from processor stack
C: loaded from processor stack

Description: Used to exit from an interrupt or TRAP service routine. The PC and PS are restored (popped) from the processor stack. If the RTI sets the T bit in the PS, a trace trap will occur prior to executing the next instruction. When executed in supervisor mode, the current and previous mode bits in the restored PS cannot be kernel. When executed in user mode, the current and previous mode bits in the restored PS can only be user. RTI cannot clear PS bit 11 if it was already set.

RTT



Operation:
PC <-- (SP) ↑
PS <-- (SP) ↑

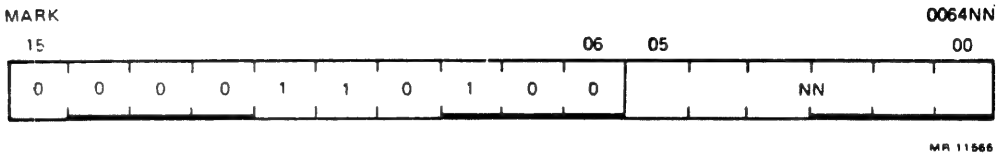
Condition Codes:
N: loaded from processor stack
Z: loaded from processor stack
V: loaded from processor stack
C: loaded from processor stack

Description: Operation is the same as RTI except that it inhibits a trace trap whereas RTI permits a trace trap. If the new PS has the T bit set,

a trap will occur after execution of the first instruction after RTT. When executed in supervisor mode, the current and previous mode bits in the restored PS cannot be kernel. When executed in user mode, the current and previous mode bits in the restored PS can only be user. RTT cannot clear PS bit 11 if it was already set.

6.3.6.6 Miscellaneous Program Control -

MARK



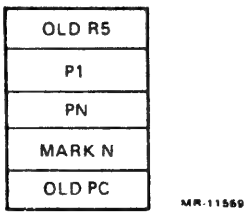
Operation:
```
SP <-- PC + 2 x NN
PC <-- R5
R5 <-- (SP)+
NN = number of parameters
```

Condition Codes:
```
N: unaffected
Z: unaffected
V: unaffected
C: unaffected
```

Description: Used as part of the standard subroutine return convention. MARK facilitates the stack clean-up procedures involved in subroutine exit. Assembler format is: MARK N.

Example:
```
MOV R5,-(SP)        ;place old R5 on stack
MOV P1,-(SP)        ;place N parameters on
MOV P2,-(SP)        ;the stack to be used
                     ;there by the subroutine
MOV PN,-(SP)
MOV =MARKN,-(SP)    ;place the instruction
                     ;MARK N on the stack
MOV SP,R5           ;set up address at MARK N
                     ;instruction
JSR PC,SUB          ;jump to subroutine
```

At this point the stack is as follows:



And the program is at the address SUB which is the beginning of the subroutine.

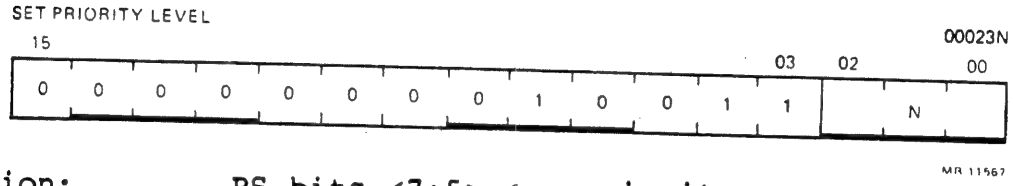
```
SUB:                    ;execution of the
                        ;subroutine itself

        RTS R5          ;the return begins:
                        ;this causes the contents
                        ;of R5 to be placed in the
                        ;PC which then results in
                        ;the execution of the
                        ;instruction MARK N.  The
                        ;contents of the old PC
                        ;are placed in R5.
```

MARK N causes: (1) the stack pointer to be adjusted to point to the old R5 value; (2) the value now in R5 (the old PC) to be placed in the PC; and (3) contents of the old R5 to be popped into R5 thus completing the return from subroutine.

NOTE

If memory management is in use, the stack must be mapped through both I and D space to execute the MARK instruction.

SPL



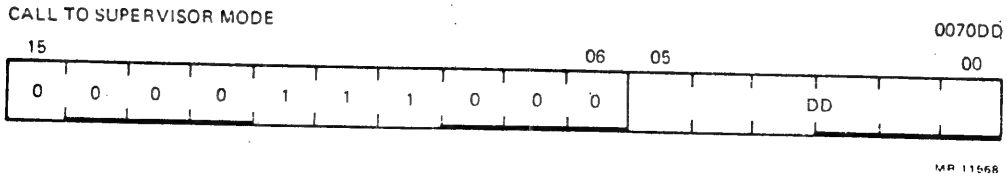
Operation: PS bits <7:5> <-- priority (priority = N)

Condition Codes:
N: unaffected
Z: unaffected
V: unaffected
C: unaffected

Description: In kernel mode, the least significant three bits of the instruction are loaded into the processor status word (PS) bits <7:5>, thus causing a changed priority. The old priority is lost. In user or supervisor modes, SPL executes as a NOP.

Assembler syntax is: SPL N

CSM



Operation:

```
If MMR3 bit 3 = 1 and current
mode = kernel then
supervisor SP <-- current mode SP
temp<15:4> <-- PS<15:4>
temp<3:0> <-- 0
PS<13:12> <-- PS<15:14>
PS<15:14> <-- 01
PS 4 <-- 0
-(SP) <-- temp
-(SP) <-- PC
-(SP) <-- (dst)
PC <-- (10)
otherwise, traps to 10 in kernel mode.
```

Condition Codes:

N: unaffected
Z: unaffected
V: unaffected
C: unaffected

Description:

CSM may be executed in user or supervisor mode, but is an illegal instruction in kernel mode. CSM copies the current stack pointer (SP) to the supervisor mode, switches to supervisor mode, stacks three words on the supervisor stack (the PS with the condition codes cleared, the PC, and the argument word addressed by the operand), and sets the PC to the contents of location 10 (in supervisor space). The called program in supervisor space may return to the calling program by popping the argument word from the stack and executing RTI. On return, the condition codes are determined by the PS word on the stack. Hence, the called program in supervisor space may control the condition code values following return.

6.3.6.7 Reserved Instruction Traps - These are caused by attempts to execute instruction codes reserved for future processor expansion (reserved instructions) or instructions with illegal addressing modes (illegal instructions). Order codes not corresponding to any of the instructions described are considered to be reserved instructions. JMP and JSR with register mode destinations are illegal instructions; they trap to virtual address 4 in kernel data space. Reserved instructions trap to vector address 10 in kernel data space.

6.3.6.8 Trace Trap - Trace trap is enabled by bit 4 of the PS and causes processor traps at the end of instruction execution. The instruction that is executed after the instruction that set the T bit will proceed to completion and then trap through the trap vector at address 14. Note that the trace trap is a system debugging aid and is transparent to the general programmer.

NOTE

Bit 4 of the PS can only be set indirectly by executing a RTI or RTT instruction with the desired PS on the stack.

6.3.6.8.1 Special Cases Of The T Bit - The following are special cases of the T bit.

NOTE

The traced instruction is the instruction after the one that set the T bit.

1. An instruction that cleared the T bit -- Upon fetching the traced instruction, an internal flag, the trace flag, was set. The trap will still occur at the end of this instruction's execution. The status word on the stack, however, will have a clear T bit.

2. An instruction that set the T bit -- Since the T bit was already set, setting it again has no effect. The trap will occur.

3. An instruction that caused an instruction trap -- The instruction trap is performed and the entire routine for the service trap is executed. If the service routine exits with an RTI, or in any other way restores the stacked status word, the T bit is set again, the instruction following the traced instruction is executed, and, unless it is one of the special cases noted previously, a trace trap occurs.

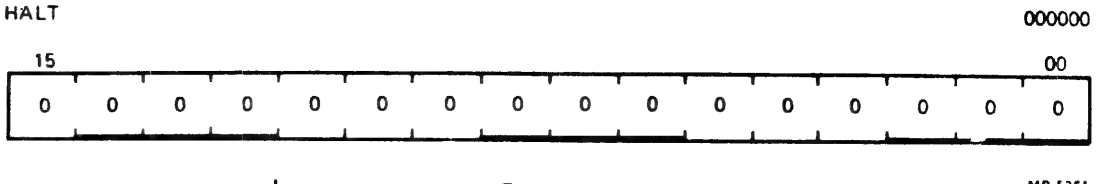
4. An instruction that caused a stack overflow -- The instruction completes execution as usual. The stack overflow does not cause a trap. The trace trap vector is loaded into the PC and PS and the old PC and PS are pushed onto the stack. Stack overflow occurs again, and this time the trap is made.

5. An interrupt between setting of the T-bit and fetch of the traced instruction -- The entire interrupt service routine is executed and then the T-bit is set again by the exiting RTI. The traced instruction is executed (if there have been no other interrupts) and, unless it is a special case noted above, causes a trace trap.

6. Interrupt trap priorities -- See Table 1-8.

6.3.7 Miscellaneous Instructions -

HALT



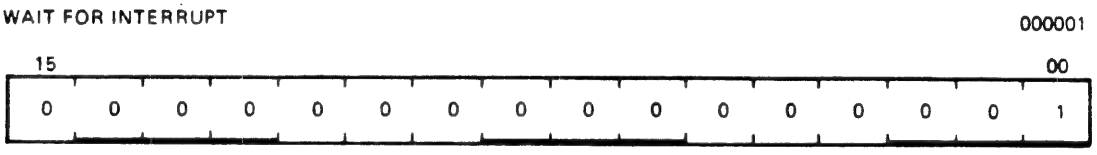
Operation:

```
↓ (SP) <-- PS
↓ (SP) <-- PC
PC <-- restart address
PS <-- 340
```

Condition Codes: Not affected

Description: The effect of HALT depends upon the CPU operating mode and the halt option currently selected. See Chapter 8 - Interfacing for more details on halt options. In kernel mode, a halt option of 1 (external logic driving a 1 on DAL3 in response to a GP Read with a GP code of 000) causes a trap through location 4 and sets bit 7 of the CPU error register when HALT is executed. If the halt option is 0 in kernel mode, execution of the HALT instruction causes the DCJ11 into console ODT. Execution of the HALT instruction in user or supervisor mode causes a trap through location 4 and sets bit 7 of the CPU error register.

WAIT

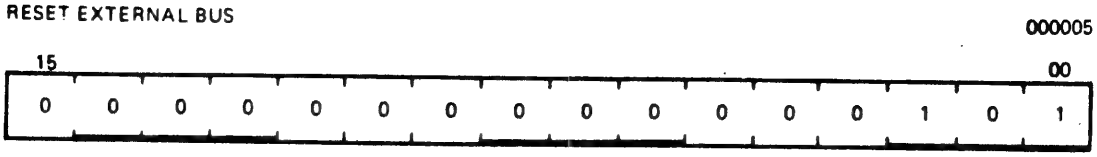


Condition Codes: Not affected

Description: In WAIT, as in all instructions, the PC points to the next instruction following the WAIT instruction. Thus, when an interrupt causes the PC and PS to be pushed onto the processor stack, the address of the next instruction following the WAIT is saved. The exit from the interrupt routine (i.e., execution of an RTI instruction) will cause resumption of the interrupted process at the instruction following the WAIT. If not in kernel mode, WAIT executes as a NOP.

RESET

RESET EXTERNAL BUS

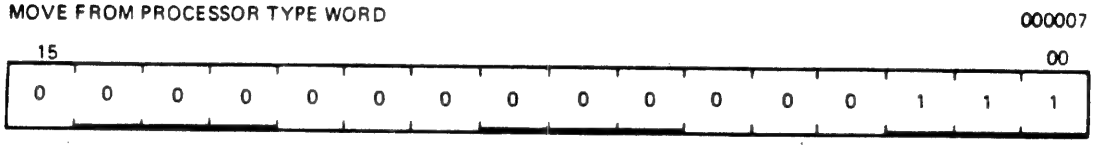


Condition Codes: Not affected

Description: The following sequence of events occurs: (1) a GP Write cycle is performed and a GP code of 014 is generated, (2) operation is delayed for 69 microcycles, (3) a GP Write is performed and a GP code of 214 is generated, (4) operation is delayed for 600 microcycles delay. If not in kernel mode, RESET operates as a NOP.

MFPT

MOVE FROM PROCESSOR TYPE WORD



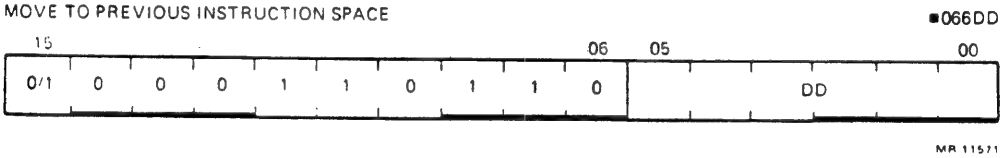
Operation: R0 <-- 5

Condition Codes: Not affected

Description: The number 5 is placed in R0, indicating to the system software that the processor type is DCJ11.

MTPD/MTPI

MOVE TO PREVIOUS DATA SPACE
MOVE TO PREVIOUS INSTRUCTION SPACE



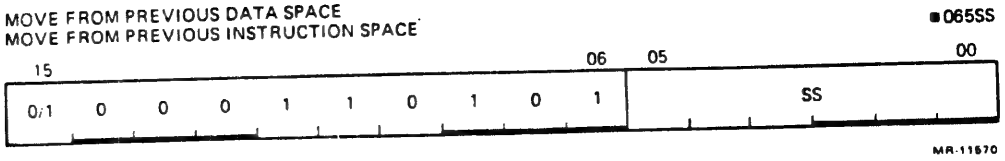
Operation: (temp) <-- (SP)+
(dst) <-- (temp)

Condition Codes: N: set if the source < 0
Z: set if the source = 0
V: cleared
Z: unaffected

Description: The instruction pops a word off the current stack determined by PS bits <15:14> and stores that word into an address in the previous space (PS bits <13:12>). The destination address is computed using the current registers and memory map.

MFPD/MFPI

MOVE FROM PREVIOUS DATA SPACE
MOVE FROM PREVIOUS INSTRUCTION SPACE



Operation:
(temp) <-- (src)
-(SP) <-- (temp)

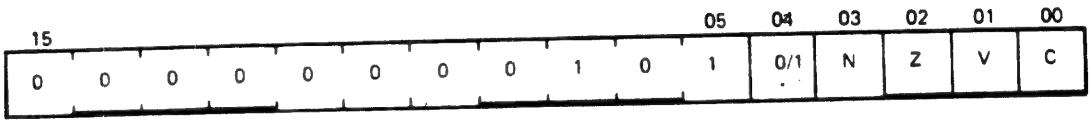
Condition Codes:
N: set if the source < 0
Z: set if the source = 0
V: cleared
Z: unaffected

Description:
Pushes a word onto the current stack from an address in the previous space determined by PS<13:12>. The source address is computed using the current registers and memory map. When MFPS is executed and both previous mode current mode are user, the instruction functions as though it were MFPD.

6.3.8 Condition Code Operators -

CLN SEN
CLZ SEZ
CLV SEV
CLC SEC
CCC SCC

CONDITION CODE OPERATORS



Description:
Set and clear condition code bits. Selectable combinations of these bits may be cleared or se together. Condition code bits corresponding to bits in the condition code operator (bits <3:0> are modified according to the sense of bit 4, the set/clear bit of the operator; i.e., set th bit specified by bit 0, 1, 2, or 3, if bit 4 = 1. Clear corresponding bits if bit 4 = 0.

Mnemonic	Operation	OP Code
CLC	Clear C	000241
CLV	Clear V	000242
CLZ	Clear Z	000244
CLN	Clear N	000250
SEC	Set C	000261
SEV	Set V	000262
SEZ	Set Z	000264
SEN	Set N	000270
SCC	Set all CCs	000277

CCC	Clear all CCs	000257
	Clear V and C	000243
NOP	No operation	000240

Combinations of the above set or clear operations may be ORed together to form combined instructions.

7.1 INTRODUCTION

The DCJ11 executes forty-six floating-point instructions. The floating-point instruction set is compatible with the FP11 instruction set for PDP-11 computers. Both single- and double-precision floating-point capabilities are available with other features, including floating-to-integer and integer-to-floating conversion.

7.2 FLOATING-POINT DATA FORMATS

Mathematically, a floating-point number may be defined as having the form (2 ** K) * f, where K is an integer and f is a fraction. For a nonvanishing number, K and f are uniquely determined by imposing the condition $1/2 \leq f < 1$. The fractional part (f) of the number is then said to be normalized. For the number 0, f is assigned the value 0, and the value of K is indeterminate.

The floating-point data formats are derived from this mathematical representation for floating-point numbers. Two types of floating-point data are provided. In single-precision, or floating mode, the data is 32 bits long. In double-precision, or double mode, the data is 64 bits long. Sign magnitude notation is used.

7.2.1 Nonvanishing Floating-Point Numbers - The fractional part (f) is assumed normalized, so that its most significant bit must be 1. This 1 is the "hidden" bit: it is not stored explicitly in the data word, but the microcode restores it before carrying out arithmetic operations. The floating and double modes reserve 23 and 55 bits, respectively, for f. These bits, with the hidden bit, imply effective word lengths of 24 bits and 56 bits.

Eight bits are reserved for storage of the exponent K in excess 200 notation (i.e., as K + 200 (octal)), giving a biased exponent. Thus, exponents from -128 to +127 could be represented by 0 to 377 (base 8), or 0 to 255 (base 10). For reasons given below, a biased exponent of 0 (the true exponent of -200 (octal)), is reserved for floating-point 0. Therefore, exponents are restricted to the range -127 to +127 inclusive (-177 to +177 octal) or, in excess 200 notation, 1 to 377.

The remaining bit of the floating-point word is the sign bit. The number is negative if the sign bit is a 1.

7.2.2 Floating-Point Zero - Because of the hidden bit, the fractional part is not available to distinguish between 0 and nonvanishing numbers whose fractional part is exactly 1/2. Therefore, the DCJ11 reserves a biased exponent of 0 for this purpose, and any floating-point number with a biased exponent of 0 either traps or is treated as if it were an exact 0 in arithmetic operations. An exact or "clean" 0 is represented by a word whose bits are all 0s. A "dirty" 0 is a floating-point number with a biased exponent of 0 and a nonzero fractional part. An arithmetic operation for which the resulting true exponent exceeds 277 (octal) is regarded as producing a floating overflow; if the true exponent is less than -177 (octal), the operation is regarded as producing a floating underflow. A biased exponent of 0 can thus arise from arithmetic operations as a special case of overflow (true exponent = -200 octal). (Recall that only eight bits are reserved for the biased exponent.) The fractional part of results obtained from such overflow and underflow is correct.

7.2.3 Undefined Variables - An undefined variable is any bit pattern with a sign bit of 1 and a biased exponent of 0. The term "undefined variable" is used, for historical reasons, to indicate that these bit patterns are not assigned a corresponding floating-point arithmetic value. Note that the undefined variable is frequently referred to as -0 elsewhere in this chapter.

A design objective was to assure that the undefined variable would not be stored as the result of any floating-point operation in a program run with the overflow and underflow interrupts disabled. This is achieved by storing an exact 0 on overflow and underflow, if the corresponding interrupt is disabled. This feature, together with an ability to detect reference to the undefined variable (implemented by the FIUV bit discussed later), is intended to provide the user with a debugging aid: if -0 occurs, it did not result from a previous floating-point arithmetic instruction.

7.2.4 Floating-Point Data - Floating-point data is stored in words of memory as illustrated in Figures 7-1 and 7-2.

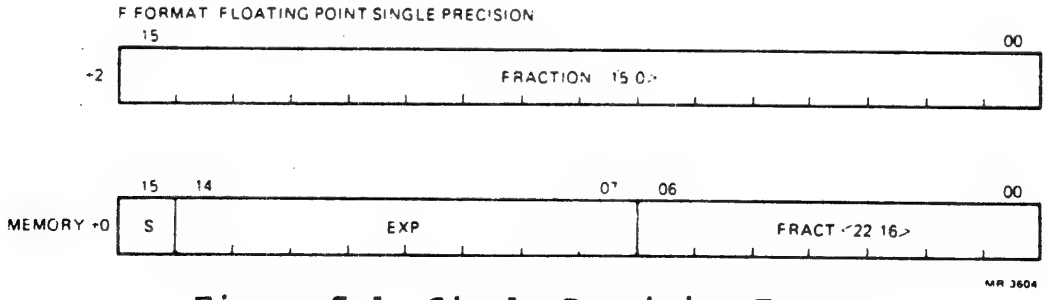


Figure 7-1 Single-Precision Format

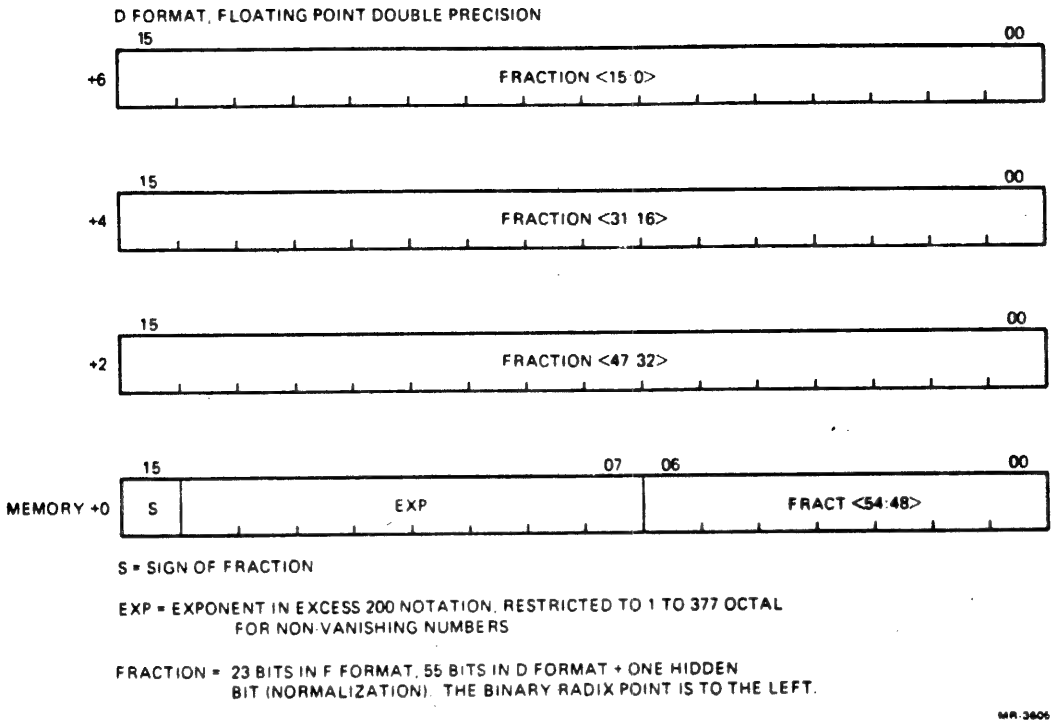


Figure 7-2 Double-Precision Format

The DCJ11 provides for conversion of floating-point to integer format and vice-versa. The processor recognizes single-precision integer (I) and double-precision integer long (L) numbers, which are stored in standard 2's complement form. (See Figure 7-3.)

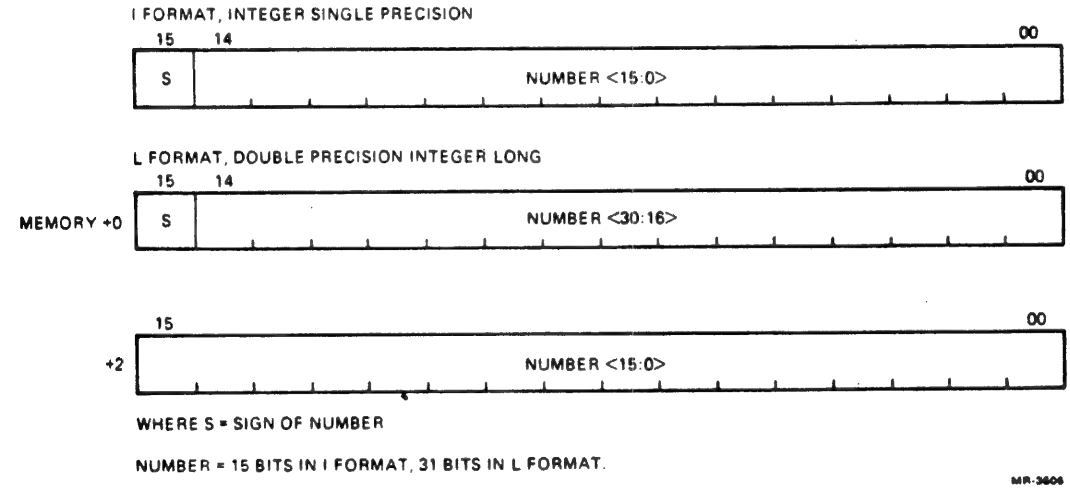


Figure 7-3 2's Complement Format

7.3 FLOATING-POINT STATUS REGISTER (FPS)

This register provides mode and interrupt control for the currently executing floating-point instruction and also reflects conditions resulting from the execution of the previous instruction. (See Figure 7-4.) In this discussion a set bit = 1 and a reset bit = 0. Three bits of the FPS register control the modes of operation.

1. Single/Double -- Floating-point numbers can be either single- or double-precision.

2. Long/Short -- Integer numbers can be 16 bits or 32 bits.

3. Chop/Round -- The result of a floating-point operation can be either "chopped" or "rounded." The term "chop" is used instead of "truncate" in order to avoid confusion with truncation of series used in approximations for function subroutines.

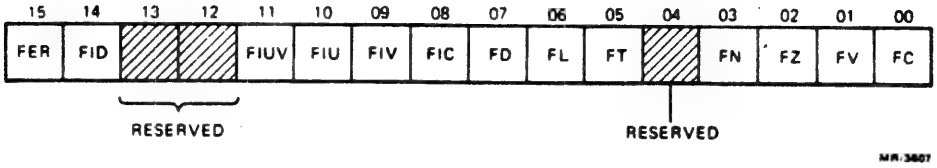


Figure 7-4 Floating-Point Status Register

The FPS register contains an error flag and four condition codes (5 bits): carry, overflow, zero, and negative, which are analogous to the CPU condition codes.

The DCJ11 recognizes six floating-point exceptions:

- o Detection of the presence of the undefined variable in memory
- o Floating overflow
- o Floating underflow
- o Failure of floating-to-integer conversion
- o Attempt to divide by 0
- o Illegal floating op code

For the first four of these exceptions, bits in the FPS register are available to individually enable and disable interrupts. An interrupt on the occurrence of either of the last two exceptions can be disabled only by setting a bit that disables interrupts on all six of the exceptions, as a group.

Of the 13 FPS bits, 5 are set as part of the output of a floating-point instruction: the error flag and condition codes. Any of the mode and interrupt control bits may be set by the user; the LDFPS instruction is available for this purpose. These thirteen bits are stored in the FPS register as shown in Figure 7-4. The FPS register bits are described in Table 7-1.

Table 7-1 FPS Register Bits

Bit	Name	Description
15	Floating Error (FER)	The FER bit is set by the DCJ11 if:

1. division by zero occurs.

2. an illegal op code occurs.

3. any one of the remaining floating point exceptions occurs and the corresponding interrupt is enabled.

Note that the above action is independent of whether the FID bit is set or clear.

Note also that the DCJ11 never resets the FER bit. Once the FER bit is set by the DCJ11, it can be cleared only by an LDFPS instruction (note the RESET instruction does not clear the FER bit). This means that the FER bit is up-to-date only if the most recent floating-point instruction produced a floating-point exception.

14	Interrupt Disable (FID)	If the FID bit is set, all floating-point interrupts are disabled.

NOTE

1. The FID bit is primarily a maintenance feature. It should normally be clear. In particular, it must be clear is one wishes to assure that storage of -0 by the DCJ11 is always accompanied by an interrupt.

2. Throughout the rest of the chapter assume that the FID bit is clear in all discussions involving overflow, underflow, occurrence of -0, and integer conversion errors.

13		Reserved for future DIGITAL use.
12		Reserved for future DIGITAL use.
11	Interrupt on Undefined Variable (FIUV)	An interrupt occurs if FIUV is set and a -0 is obtained from memory as an operand of ADD, SUB, MUL, DIV, CMP, MOD, NEG, ABS, TST, or any LOAD instruction. The interrupt occurs before execution on all instructions. When FIUV is reset, -0 can be loaded and used in any floating-point operation. Note that the interupt is not activated by the presence of -0 in an AC operand of an arithmetic instruction; in particular, trap on

		-0 never occurs in mode 0. A result of -0 will not be stored without the simultaneous occurrence of an interrupt.
10	Interrupt on Underflow (FIU)	When the FIU bit is set, floating underflow will cause an interrupt. The fractional part of the result of the operation causing the interrupt will be correct. The biased exponent will be too large by 400, except for the special case of 0, which is correct. An exception is discussed later in the detailed description of the LDEXP instruction.
9	Interrupt on Overflow (FIV)	When the FIV bit is set, floating overflow will cause an interrupt. The fractional part of the result of the operation causing the overflow will be correct. The biased exponent will be too small by 400. If the FIV bit is reset and overflow occurs, there is no interrupt. The DCJ11 returns exact 0. Special cases of overflow are discussed in the detailed descriptions of the MOD and LDEXP instructions.
8	Interrupt on Integer Conversion Error (FIC)	When the FIC bit is set and a conversion to integer instruction fails, an interrupt will occur. If the interrupt occurs, the destination is set to 0, and all other registers are left untouched. If the FIC bit is reset, the result of the operation will be the same as detailed above, but no interrupt will occur. The conversion instruction fails if it generates an integer with more bits than can fit in the short or long integer word specified by the FL bit.
7	Floating Double-Precision Mode (FD)	The FD bit determines the precision that is used for floating-point calculations. When set, double-precision is assumed; when reset, single-precision is used.

6	Floating Long-Integer Mode (FL)	The FL bit is active in conversion between integer and floating-point formats. When set, the integer format assumed is double-precision 2's complement (i.e., 32 bits). When reset, the integer format is assumed to be single-precision 2's complement (i.e., 16 bits).
5	Floating Chop Mode (FT)	When the FT bit is set, the result of any arithmetic operation is chopped (truncated). When reset, the result is rounded.
4		Reserved for future DIGITAL use.
3	Floating Negative (FN)	FN is set if the result of the last floating-point operation was negative; otherwise it is reset.
2	Floating Zero (FZ)	FZ is set if the result of the last floating-point operation was 0; otherwise it is reset.
1	Floating Overflow (FV)	FV is set if the last floating-point operation resulted in an exponent overflow; otherwise it is reset.
0	Floating Carry (FC)	FC is set if the last floating-point operation resulted in a carry of the most significant bit. This can only occur in floating double-to-integer conversions.

7.4 FLOATING EXCEPTION CODE AND ADDRESS REGISTERS

One interrupt vector is assigned to take care of all floating-point exceptions (location 244). The six possible errors are coded in the 4-bit floating exception code (FEC) register as follows.

2	Floating op-code error
4	Floating divide by zero
6	Floating-to-integer or double-to-integer conversion error
8	Floating overflow
10	Floating underflow
12	Floating undefined variable

The address of the instruction producing the exception is stored in the floating exception address (FEA) register.

The FEC and FEA registers are updated only when one of the

following occurs.

1. Division by zero.

2. Illegal op code.

3. Any of the other four exceptions with the corresponding interrupt enabled.

This implies that only when the FER bit is set are the FEC and FEA registers updated.

NOTE

1. If one of the last four exceptions occurs with the corresponding interrupt disabled, the FEC and FEA are not updated.

2. If an exception occurs, inhibition of interrupts by the FID bit does not inhibit updating of the FEC and FEA.

3. The FEC and FEA are not updated if no exception occurs. This means that the STST (store status) instruction will return current information only if the most recent floating-point instruction produced an exception.

4. Unlike the FPS, no instructions are provided for storage into the FEC and FEA registers.

7.5 FLOATING-POINT INSTRUCTION ADDRESSING

Floating-point instructions use the same type of addressing as the central processor instructions. A source or destination operand is specified by designating one of eight addressing modes and one of eight central processor general registers to be used in the specified mode. The modes of addressing are the same as those of the central processor, except in mode 0. In mode 0 the operand is located in the designated floating-point processor accumulator rather than in a central processor general register. The modes of addressing are as follows.

0 = Floating-point accumulator
1 = Deferred
2 = Autoincrement
3 = Autoincrement-deferred
4 = Autodecrement
5 = Autodecrement-deferred
6 = Indexed
7 = Indexed-deferred

Autoincrement and autodecrement operate on increments and decrements of 4 for F format and 10 (octal) for D format.

In mode 0 users can make use of all six floating-point accumulators (AC0 - AC5) as their source or destination. Specifying floating-point accumulators AC6 or AC7 will result in an illegal op code trap. In all other modes, which involve transfer of data to or from memory or the general registers, users are restricted to the first four floating-point accumulators (AC0 - AC3). When reading or writing a floating-point number from or to memory, the low memory word contains the most significant word of the floating-point number, and the high memory word the least significant word.

7.6 ACCURACY

General comments on the accuracy of the DCJ11 floating-point instructions are presented here. The descriptions of the individual instructions include the accuracy at which they operate. An instruction or operation is regarded as "exact" if the result is identical to an infinite precision calculation involving the same operands. The a priori accuracy of the operands is thus ignored. All arithmetic instructions treat an operand whose biased exponent is 0 as an exact 0 (unless FIUV is enabled and the operand is -0, in which case an interrupt occurs). For all arithmetic operations, except DIV, a 0 operand implies that the instruction is exact. The same statement holds for DIV if the 0 operand is the dividend. But if it is the divisor, division is undefined and an interrupt occurs.

For nonvanishing floating-point operands, the fractional part is binary normalized. It contains 24 bits or 56 bits for floating mode and double mode, respectively. For ADD, SUB, MUL, and DIV, two guard bits are necessary and sufficient for the general case to guarantee return of a chopped or rounded result identical to the corresponding infinite precision operation chopped or rounded to the specified word length. Thus, with two guard bits, a chopped result has an error bound of one least significant bit (LSB); a rounded result has an error bound of 1/2 LSB. These error bounds are realized by the DCJ11 of all instructions.

In the rest of this chapter, an arithmetic result is called exact if no nonvanishing bits would be lost by chopping. The first bit lost in chopping is referred to as the "rounding" bit. The value of a rounded result is related to the chopped result as follows.

1. If the rounding bit is 1, the rounded result is the chopped result incremented by an LSB.

2. If the rounding bit is 0, the rounded and chopped results are identical.

It follows that:

1. If the result is exact: rounded value = chopped value = exact value.

2. If the result is not exact, its magnitude is:

   - o always decreased by chopping.
   - o decreased by rounding if the rounding bit is 0.
   - o increased by rounding if the rounding bit is 1.

Occurrence of floating-point overflow and underflow is an error condition: the result of the calculation cannot be correctly stored because the exponent is too large to fit into the eight bits reserved for it. However, the internal hardware has produced the correct answer. For the case of underflow, replacement of the correct answer by 0 is a reasonable resolution of the problem for many applications. This is done by the DCJ11 if the underflow interrupt is disabled. The error incurred by this action is an absolute rather than a relative error; it is bounded (in absolute value) by 2 ** -128. There is no such simple resolution for the case of overflow. The action taken, if the overflow interrupt is disabled, is described under FIV (bit 9) in Table 7-1.

The FIV and FIU bits (of the floating-point status word) provide users with an opportunity to implement their own correction of an overflow or underflow condition. If such a condition occurs and the corresponding interrupt is enabled, the microcode stores the fractional part and the low eight bits of the biased exponent. The interrupt will take place and users can identify the cause by examination of the FV (floating overflow) bit or the FEC (floating exception) register. The reader can readily verify that (for the standard arithmetic operations ADD, SUB, MUL, and DIV) the biased exponent returned by the instruction bears the following relation to the correct exponent.

1. On overflow, it is too small by 400 (octal)

2. On underflow, if the biased exponent is 0, it is correct. If the biased exponent is not 0, it is too large by 400 (octal).

Thus, with the interrupt enable, enough information is available to determine the correct answer. Users may, for example, rescale their variables (via STEXP and LDEXP) to continue a calculation. Note that the accuracy of the fractional part is unaffected by the occurrence of underflow or overflow.

7.7 FLOATING-POINT INSTRUCTIONS

Each instruction that references a floating-point number can operate on either single- or double-precision numbers, depending on the state of the FD mode bit. Similarly, there is a mode bit FL that determines whether a 32-bit integer (FL = 1) or a 16-bit integer (FL = 0) is used in conversion between integer and floating-point representations. FSRC and FDST operands use floating-point addressing modes (see Figure 7-5); SRC and DST operands use CPU addressing modes.

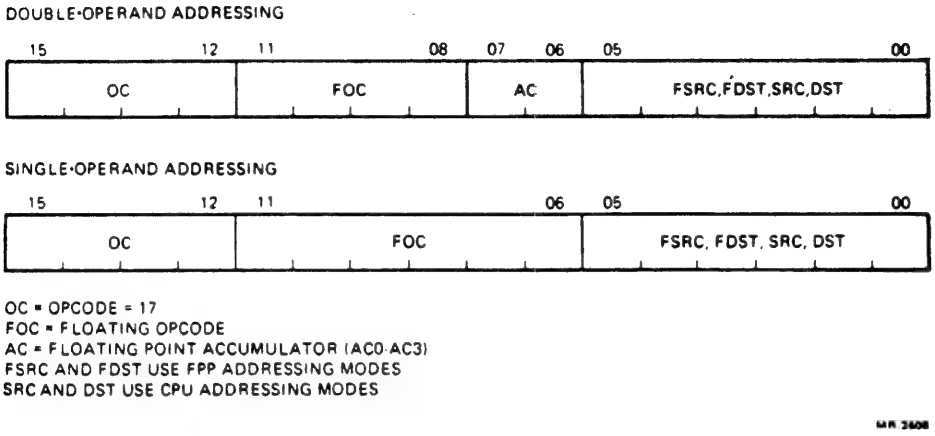


Figure 7-5 Floating-Point Addressing Modes

Terms Used in Instruction Definitions

OC = opcode = 17

FOC = floating opcode

AC = contents of accumulator, as specified by AC field of instruction.

fsrc = address of floating-point source operand

fdst = address of floating-point destination operand

f = fraction

XL = largest fraction that can be represented:

```
1 - 2 ** (-24), FD = 0; single-precision
1 - 2 ** (-56), FD = 1; double-precision
```

XLL = smallest number that is not identically zero =

```
2 ** (-128)
```

XUL = largest number that can be represented =

```
2 ** (127) * XL
```

JL = largest integer that can be represented:

2 ** (15) - 1; FL = 0; short integer
2 ** (31) - 1; FL = 1; long integer

ABS (address) = absolute value of (address)

EXP (address) = biased exponent of (address)

.LT. = "less than"

.LE. = "less than or equal to"

.GT. = "greater than"

.GE. = "greater than or equal to"

LSB = least significant bit

Boolean Symbols

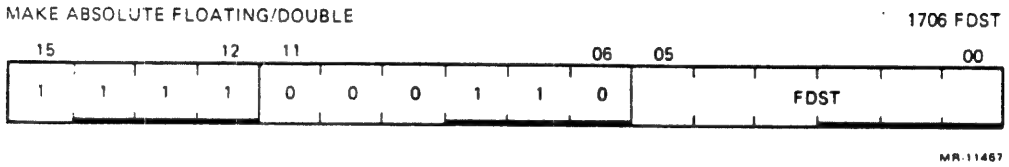
$\wedge$ = AND

$\vee$ = inclusive OR

$\not\vee$ = exclusive OR

~ = NOT

ABSF/ABSD



Format: ABSF FDST

Operation: If (FDST) < 0, (FDST) <-- -(FDST).

If EXP(FDST) = 0, (FDST) <-- exact 0.

For all other cases, (FDST) <-- (FDST).

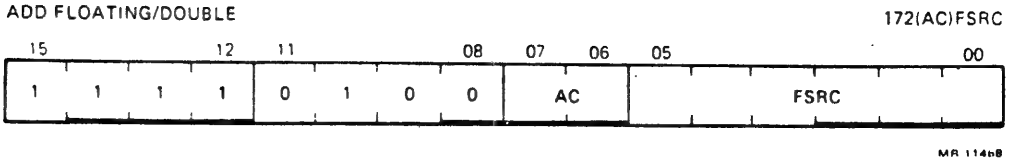
Condition Codes:
FC <-- 0
FV <-- 0
FZ <-- 1 if (FDST) = 0, else FZ <-- 0
FN <-- 0

Description: Set the contents of FDST to its absolute value.

Interrupts: If FIUV is enabled, trap on -0 occurs before execution. Overflow and underflow cannot occur.

Accuracy: These instructions are exact.

ADDF/ADDD



Format: ADDF FSRC,AC

Operation: Let SUM = (AC) + (FSRC)

If underflow occurs and FIU is not enabled, AC <-- exact 0.

If overflow occurs and FIV is not enabled, AC <-- exact 0.

For all others cases, AC <-- SUM.

Condition Codes:
FC <-- 0
FV <-- 1 if overflow occurs, else FV <-- 0
FZ <-- 1 if (AC) = 0, else FZ <-- 0
FN <-- 1 if (AC) < 0, else FN <-- 0

Description: Add the contents of FSRC to the contents of AC. The addition is carried out in single- or double-precision and is rounded or chopped in accordance with the values of the FD and FT bits in the FPS register. The result is stored in AC except for:

1. Overflow with interrupt disabled.
2. Underflow with interrupt disabled.

For these exceptional cases, an exact 0 is stored in AC.

Interrupts: If FIUV is enabled, trap on -0 in FSRC occurs before execution. If overflow or underflow occurs, and if the corresponding interrupt is enabled, the trap occurs with the faulty result in AC. The fractional parts are correctly stored. The exponent part is too small by 400 for overflow. It is too large by 400 for underflow, except for the special case of 0, which is correct.

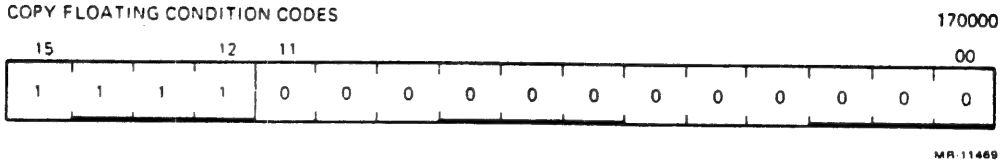
Accuracy: Errors due to overflow and underflow are described above. If neither occurs, then: for oppositely signed operands with exponent difference of 0 or 1, the answer returned is exact if a loss of significance of one or more bits can occur. Note that these are the only cases for which loss of significance of more than one bit can occur. For all other cases the

result is inexact with error bounds of:

1. LSB in chopping mode with either single- or double-precision.
2. 1/2 LSB in rounding mode with either single- or double-precision.

Special Comment: The undefined variable -0 can occur only in conjunction with overflow or underflow. It will be stored in AC only if the corresponding interrupt is enabled.

CFCC

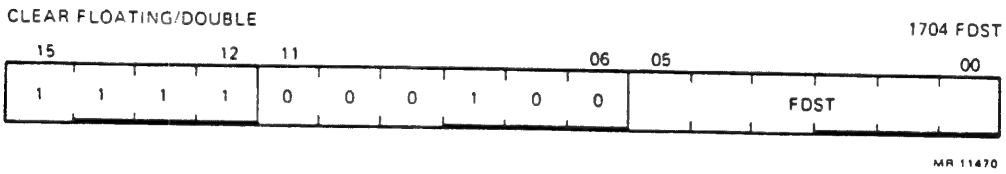


Format: CFCC

Operation:
C <-- FC
V <-- FV
Z <-- FZ
N <-- FN

Description: Copy the floating-point condition codes into the CPU's condition codes.

CLRF/CLRD



Format: CLRF FDST

Operation: (FDST) <-- exact 0

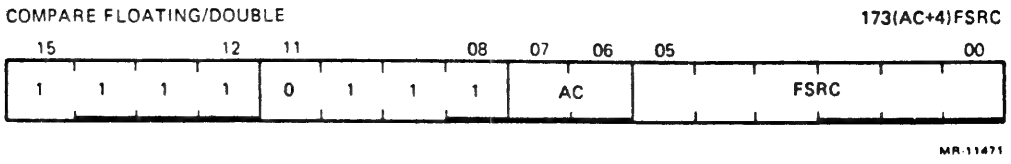
Condition Codes:
FC <-- 0
FV <-- 0
FZ <-- 1
FN <-- 0

Description: Set FDST to 0. Set FZ condition code and clear other condition code bits.

Interrupts: No interrupts will occur. Overflow and underflow cannot occur.

Accuracy: These instructions are exact.

CMPF/CMPD



Format: CMPF FSRC,AC

Operation: (FSRC) - (AC)

Condition Codes:
FC <-- 0
FV <-- 0
FZ <-- 1 if (FSRC) = 0, else FZ <-- 0
FN <-- 1 if (FSRC) < 0, else FN <-- 0

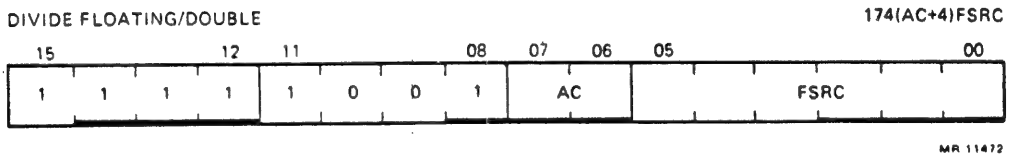
Description: Compare the contents of FSRC with the accumulator. Set the appropriate floating-point condition codes. FSRC and the accumulator are left unchanged except as noted below.

Interrupts: If FIUV is enabled, trap on -0 occurs before execution.

Accuracy: These instructions are exact.

Special Comment: An operand that has a biased exponent of 0 is treated as if it were an exact 0. In this case, where both operands are 0, the DCJ11 will store an exact 0 in AC.

DIVF/DIVD



Format: DIVF FSRC,AC

Operation: If EXP(FSRC) = 0, (AC) <-- (AC) and the instruction is aborted.

If EXP(AC) = 0, (AC) <-- exact 0.

For all other cases, let QUOT = (AC)/(FSRC).

If underflow occurs and FIU is not enabled, AC <-- exact 0.

If overflow occurs and FIV is not enabled, AC <-- exact 0.

For all others cases, AC <-- QUOT.

Condition Codes:
FC <-- 0
FV <-- 1 if overflow occurs, else FV <-- 0
FZ <-- 1 if (AC) = 0, else FZ <-- 0
FN <-- 1 if (AC) < 0, else FN <-- 0

Description: If either operand has a biased exponent of 0, it is treated as an exact 0. For FSRC this would imply division by 0; in this case the instruction is aborted, the FEC register is set to 4, and an interrupt occurs. Otherwise, the quotient is developed to single- or double-precision with two guard bits for correct rounding. The quotient is rounded or chopped in accordance with the values of the FD and FT bits in the FPS register. The result is stored in the AC except for:

1. Overflow with interrupt disabled.
2. Underflow with interrupt disabled.

For these exceptional cases, an exact 0 is stored in AC.

Interrupts: If FIUV is enabled, trap on -0 in FSRC occurs before execution. If (FSRC) = 0, interrupt traps on an attempt to divide by 0. If overflow or underflow occurs, and if the corresponding interrupt is enabled, the trap occurs with the faulty result in AC. The fractional parts are correctly stored. The exponent part is too small by 400 for overflow. It is too large by 400 for underflow, except for the special case of 0, which is correct.

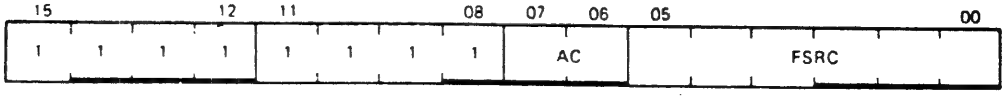
Accuracy: Errors due to overflow and underflow are described above. If none of these occurs, the error in the quotient will be bounded by 1 LSB in chopping mode and by 1/2 LSB in rounding mode.

Special Comment: The undefined variable -0 can occur only in conjunction with overflow or underflow. It will be stored in AC only if the corresponding interrupt is enabled.

LDCDF/LDCFD

LOAD AND CONVERT FROM DOUBLE-TO-FLOATING
AND FROM FLOATING-TO-DOUBLE

177(AC+4)FSRC



MR-11473

Format: LDCDF FSRC,AC

Operation: If EXP(FSRC) = 0, AC <-- exact 0.

If FD = 1, FT = 0, FIV = 0 and rounding causes overflow, AC <-- exact 0.

In all other cases, AC <-- Cxy(FSRC), where Cxy specifies conversion from floating mode x to floating mode y.

x = D, y = F if FD = 0 (single) LDCDF
y = F, y = D if FD = 1 (double) LDCFD

Condition Codes:
FC <-- 0
FV <-- 1 if conversion produces overflow, else FV <-- 0
FZ <-- 1 if (AC) = 0, else FZ <-- 0
FN <-- 1 if (AC) < 0, else FN <-- 0

Description: If the current mode is floating mode (FD = 0), the source is assumed to be a double-precision number and is converted to single-precision. If the floating chop bit (FT) is set, the number is chopped; otherwise, the number is rounded.

If the current mode is double mode (FD = 1), the source is assumed to be a single-precision number and is loaded left-justified in AC. The lower half of AC is cleared.

Interrupts: If FIUV is enabled, trap on -0 occurs before execution. Overflow cannot occur for LDCFD.

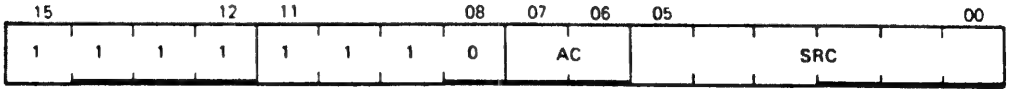
A trap occurs if FIV is enabled, and if rounding with LDCDF causes overflow. AC <-- overflowed result. This result must be +0 or -0. Underflow cannot occur.

Accuracy: LDCFD is an exact instruction. Except for overflow, described above, LDCDF incurs an error bounded by 1 LSB in chopping mode and by 1/2 LSB in rounding mode.

LDCIF/LDCID/LDCLF/LDCLD

LOAD AND CONVERT INTEGER OR LONG INTEGER
TO FLOATING OR DOUBLE-PRECISION

177(AC)SRC



MR-11474

Format: LDCIF SRC,AC

Operation: AC <-- Cjx(SRC), where Cjx specifies conversion from integer mode j to floating mode y.

j = I if FL = 0, j = L if FL = 1
x = F if FD = 0, x = D if FD = 1

Condition Codes:
FC <-- 0
FV <-- 0
FZ <-- 1 if (AC) = 0, else FZ <-- 0
FN <-- 1 if (Ac) < 0, else FN <-- 0

Description: Conversion is performed on the contents of SRC from a 2's complement integer with precision j to a floating-point number of precision x. Note that j and x are determined by the state of the mode bits FL and FD.

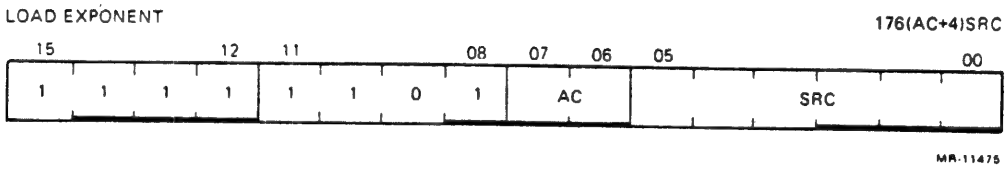
If a 32-bit integer is specified (L mode) and (SRC) has an addressing mode of 0 or immediate addressing mode is specified, the 16 bits of the source register are left-justified and the remaining 16 bits loaded with 0s before conversion.

In the case of LDCLF, the fractional part of the floating-point representation is chopped or rounded to 24 bits for FT = 1 or 0, respectively.

Interrupts: None; SRC is not floating-point, so trap on -0 cannot occur.

Accuracy: LDCIF, LDCID, and LDCLD are exact instructions. The error incurred by LDCLF is bounded by 1 LSB in chopping mode and by 1/2 LSB in rounding mode.

LDEXP



Format: LDEXP SRC,AR

Operation: NOTE: 177 and 200, appearing below, are octal numbers.

If -200 < SRC < 200, EXP(AC) <-- SRC + 200 and the rest of AC is unchanged.

If (SRC) > 177 and FIV is enabled, EXP(AC) <-- [(SRC) + 200]<7:0>.

If (SRC) > 177 and FIV is disabled, AC <-- exact 0.

If (SRC) < -177 and FIU is enabled, EXP(AC) <-- [(SRC) + 200]<7:0>.

If (SRC) < -177 and FIU is disabled, AC <-- exact 0.

Condition Codes:
FC <-- 0
FV <-- 1 if (SRC) > 177, else FV <-- 0
FZ <-- 1 if (AC) = 0, else FZ <-- 0
FN <-- 1 if (AC) < 0, else FN <-- 0

Description: Change AC so that its unbiased exponent = (SRC). That is, convert (SRC) from 2's complement to excess 200 notation and insert it into the EXP field of AC. This is a meaningful operation only if ABS(SRC) LE 177.

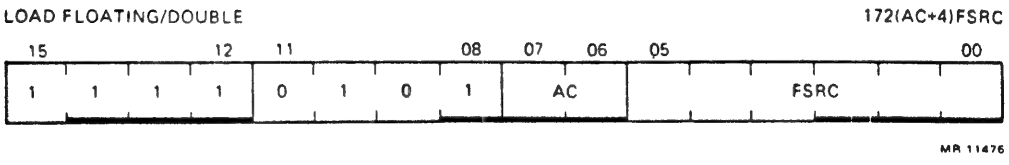
If SRC > 177, the result is treated as overflow. If SRC < -177, the result is treated as underflow.

Interrupts: No trap on -0 in AC occurs, even if FIUV is enabled. If SRC > 177 and FIV is enabled, trap on overflow will occur. If SRC < -177 and FIU is enabled, trap on underflow will occur.

Accuracy: Errors due to overflow and underflow are described above. If EXP(AC) = 0 and (SRC) = -200, AC changes from a floating-point number treated as 0 by all floating arithmetic operations to a non-0 number. This happens because the insertion of the "hidden" bit in the microcode implementation of arithmetic instructions is triggered by a nonvanishing value of EXP.

For all other cases, LDEXP implements exactly the transformation of a floating-point number (2 ** K) * f into (2 ** (SRC)) * f where 1/2 .LE. ABS(f) .LT. 1.

LDF/LDD



Format: LDF FSRC,AC

Operation: AC <-- (FSRC)

Condition Codes:
FC <-- 0
FV <-- 0

FZ <-- 1 if (AC) = 0, else FZ <-- 0
FN <-- 1 if (AC) < 0, else FN <-- 0

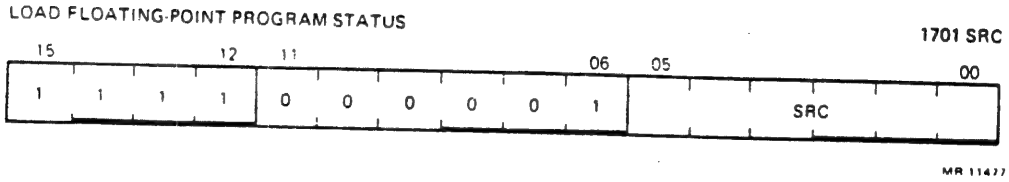
Description: Load single- or double-precision number into AC.

Interrupts: If FIUV is enabled, trap on -0 occurs before AC is loaded. Overflow and underflow cannot occur.

Accuracy: These instructions are exact.

Special Comment: These instructions permit use of -0 in a subsequent floating-point instruction if FIUV is not enabled and (FSRC) = -0.

LDFPS



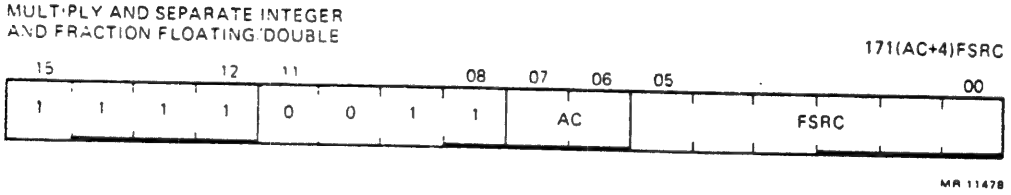
Format: LDFPS SRC

Operation: FPS <-- (SRC)

Description: Load floating-point status register from SRC.

Special Comment: Users are cautioned not to use bits 13, 12, and 4 for their own purposes, since these bits are not recoverable by the STFPS instruction.

MODF/MODD



Format: MODF FSRC,AC

Description and Operation: This instruction generates the product of its two floating-point operands, separates the product into integer and fractional parts, and then stores one or both parts as floating-point numbers.

Let PROD = (AC) * (FSRC) so that in

Floating-point: ABS(PROD) = (2 ** K) * f, where

1/2 .LE. f .LT. 1, and
EXP(PROD) = (200 + K)

Fixed-point binary: PROD = N + g, where

N = INT(PROD) = integer part of PROD, and

g = PROD - INT(PROD) = fractional part of PROD with 0 .LE. g .LT. 1.

Both N and g have the same sign as PROD. They are returned as follows:

If AC is an even-numbered accumulator (0 or 2), N is stored in AC+1 (1 or 3), and g is stored in AC.

If AC is an odd-numbered accumulator, N is not stored and g is stored in AC.

The two statements above can be combined as follows:

N is returned to AC $\vee$ 1 and g is returned to AC.

Five special cases occur, as indicated in the following formal description with L = 24 for floating mode and L = 56 for double mode.

1. If PROD overflows and FIV is enabled, AC $\vee$ 1 <-- N, chopped to L bits, AC <-- exact 0.

   Note that EXP(N) is too small by 400 and that -0 can be stored in AC $\vee$ 1.

   If FIV is not enabled, AC $\vee$ 1 <-- exact 0, AC <-- exact 0, and -0 will never be stored.

2. If 2 ** L .LE. ABS(PROD) and no overflow, AC $\vee$ 1 <-- N, chopped to L bits, AC <-- exact 0.

   The sign and EXP of N are correct, but low-order bit information is lost.

3. If 1 .LE. ABS(PROD) .LT. 2 ** L, AC $\vee$ 1 <-- N, AC <-- g.

   The integer part N is exact. The fractional part g is normalized, and chopped or rounded in accordance with FT. Rounding may cause a return of + unity for the fractional part. For L = 24, the error in g is bounded by 1 LSB in chopping mode and by 1/2 LSB in rounding mode. For L = 56, the error in g increases from the above limits as ABS(N) increases above 8 because only 59 bits of

PROD are generated.

If 2 ** p .LE. ABS(N) .LT. 2 ** (p + 1), with p > 2, the low order p - 2 bits of g may be in error.

4. If ABS(PROD) .LT. 1 and no underflow, AC $\vee$ 1 <-- exact 0 and AC <-- g.

   There is no error in the integer part. The error in the fractional part is bounded by 1 LSB in chopping mode and 1/2 LSB in rounding mode. Rounding may cause a return of + unity for the fractional part.

5. If PROD underflows and FIU is enabled, AC $\vee$ 1 <-- exact 0 and AC <-- g.

   Errors are as in case 4, except that EXP(AC) will be too large by 4008 (if EXP = 0, it is correct). Interrupt will occur and -0 can be stored in AC.

   If FIU is not enabled, AC $\vee$ 1 <-- exact 0 and AC <-- exact 0.

   For this case the error in the fractional part is less than 2 ** (-128).

Condition Codes:
FC <-- 0
FV <-- 1 if PROD overflows, else FV <-- 0
FZ <-- 1 if (AC) = 0, else FZ <-- 0
FN <-- 1 if (AC) < 0, else FN <-- 0

Interrupts: If FIUV is enabled, trap on -0 in FSRC occurs before execution. Overflow and underflow are discussed above.

Accuracy: Discussed above.

Applications:

1. Binary-to-decimal conversion of a proper fraction. The following algorithm, using MOD, will generate decimal digits D(1), D(2) . . . from left to right.

```
Initialize: I <-- 0;
            X <-- number to be converted;
            ABS(X) < 1;
While X ≠ 0 do
Begin PROD <-- X * 10;
I <-- I + 1;
D(I) <-- INT(PROD);
X <-- PROD - INT(PROD);
End;
```

   This algorithm is exact. It is case 3 in the description because the number of nonvanishing bits in the fractional part of

PROD never exceeds L, and hence neither chopping nor rounding can introduce error.

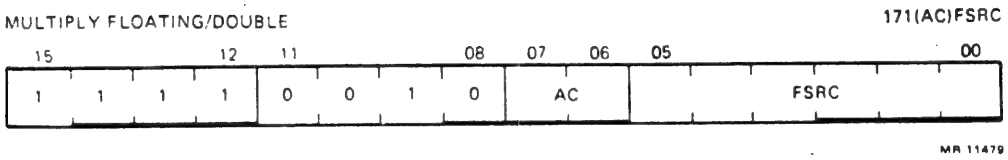
2. To reduce the argument of a trigonometric function.

   ARG * 2/PI = N + g. The low two bits of N identify the quadrant, and g is the argument reduced to the first quadrant. The accuracy of N + g is limited to L bits because of the factor 2/PI. The accuracy of the reduced argument thus depends on the size of N.

3. To evaluate the exponential function e ** x, obtain x * (log e base 2) = N + g, then e ** x = (2 ** N) * (e ** (g * ln 2)).

   The reduced argument is g * ln2 < 1 and the factor 2 ** N is an exact power of 2, which may be scaled in at the end via STEXP, ADD N to EXP and LDEXP. The accuracy of N + g is limited to L bits because of the factor (log e base 2). The accuracy of the reduced argument thus depends on the size of N.

MULF/MULD



Format: MULF FSRC,AC

Operation: Let PROD = (AC) * (FSRC)

If underflow occurs and FIU is not enabled, AC <-- exact 0.

If overflow occurs and FIV is not enabled, AC <-- exact 0.

For all others cases, AC <-- PROD.

Condition Codes:
FC <-- 0
FV <-- 1 if overflow occurs, else FV <-- 0
FZ <-- 1 if (AC) = 0, else FZ <-- 0
FN <-- 1 if (AC) < 0, else FN <-- 0

Description: If the biased exponent of either operand is 0, (AC) <-- exact 0. For all other cases PROD is generated to 48 bits for floating mode and 59 bits for double mode. The product is rounded or chopped for FT = 0 or 1, respectively, and is stored in AC except for:

1. Overflow with interrupt disabled.
2. Underflow with interrupt disabled.

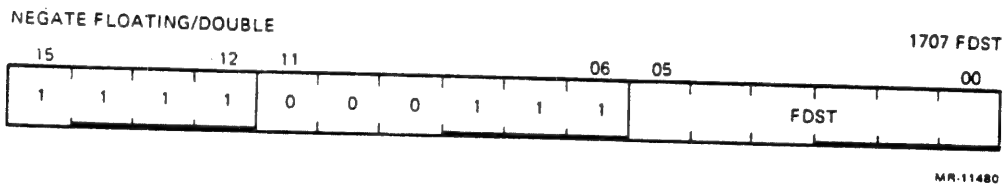
For these exceptional cases, an exact 0 is stored in AC.

Interrupts: If FIUV is enabled, trap on -0 in FSRC occurs before execution. If overflow or underflow occurs, and if the corresponding interrupt is enabled, the trap occurs with the faulty result in AC. The fractional parts are correctly stored. The exponent part is too small by 400 for overflow. It is too large by 400 for underflow, except for the special case of 0, which is correct.

Accuracy: Errors due to overflow and underflow are described above. If neither occurs, the error incurred is bounded by 1 LSB in chopping mode and 1/2 LSB in rounding mode.

Special Comment: The undefined variable -0 can occur only in conjunction with overflow or underflow. It will be stored in AC only if the corresponding interrupt is enabled.

NEGF/NEGD



Format: NEGF FDST

Operation: (FDST) <-- - (FDST) if (FDST) = 0, else
(FDST) <-- exact 0

Condition Codes: FC <-- 0
FV <-- 0
FZ <-- 1 if (FDST) = 0, else FZ <-- 0
FN <-- 1 if (FDST) < 0, else FN <-- 0

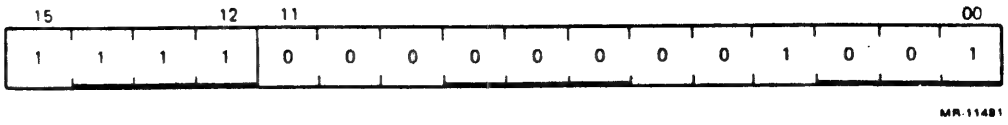
Description: Negate the single- or double-precision number; store result in same location (FDST).

Interrupts: If FIUV is enabled, trap on -0 occurs before execution. Overflow and underflow cannot occur.

Accuracy: These instructions are exact.

SETD

SET FLOATING DOUBLE MODE 170011



MR-11481

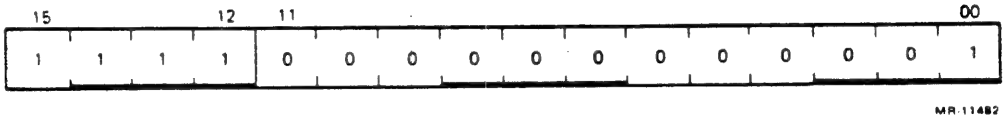
Format: SETD

Operation: FD <-- 1

Description: Set the DCJ11 in double precision mode.

SETF

SET FLOATING MODE 170001



MR-11482

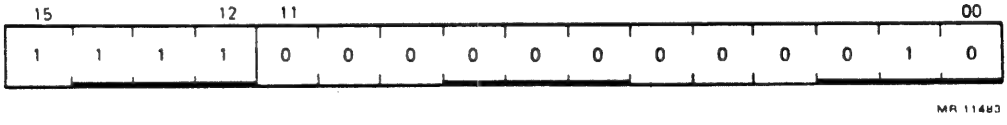
Format: SETF

Operation: FD <-- 0

Description: Set the DCJ11 in single-precision mode.

SETI

SET INTEGER MODE 177002



MR-11483

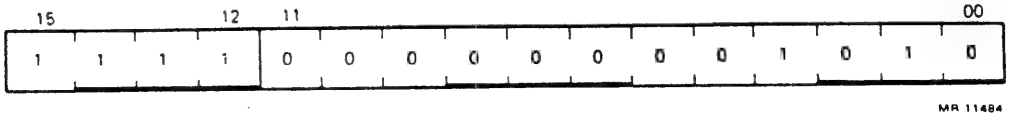
Format: SETI

Operation: FL <-- 0

Description: Set the DCJ11 for short-integer data.

SETL

SET LONG-INTEGER MODE 177012



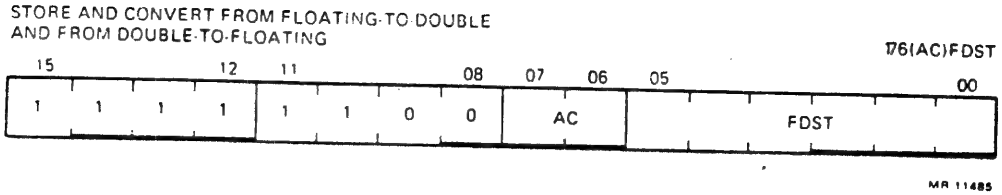
MR-11484

Format: SETL

Operation: FL <-- 1

Description: Set the DCJ11 for long-integer data.

STCFD/STCDF



Format: STCFD AC,FDST

Operation: If (AC) = 0, (FDST) <-- exact 0.

If FD = 1, FT = 0, FIV = 0 and rounding causes overflow, (FDST) <-- exact 0.

In all other cases, (FDST) <-- Cxy(AC), where Cxy specifies conversion from floating mode x to floating mode y.

x = F, y = D if FD = 0 (single) STCFD
x = D, y = F if FD = 1 (double) STCDF

Condition Codes: FC <-- 0
FV <-- 1 if conversion produces overflow, else FV <-- 0
FZ <-- 1 if (AC) = 0, else FZ <-- 0
FN <-- 1 if (AC) < 0, else FN <-- 0

Description: If the current mode is single-precision, the accumulator is stored left-justified in FDST and the lower half is cleared.

If the current mode is double-precision, the contents of the accumulator are converted to single-precision, chopped or rounded depending on the state of FT, and stored in FDST.

Interrupts: Trap on -0 will not occur even if FIUV is enabled because FSRC is an accumulator. Underflow cannot occur. Overflow cannot occur for STCFD.

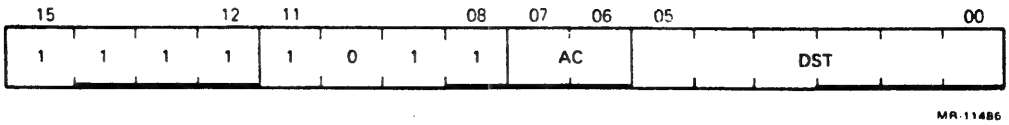
A trap occurs if FIV is enabled, and if rounding with STCDF causes overflow. (FDST) <-- overflowed result. This must be +0 or -0.

Accuracy: STCFD is an exact instruction. Except for overflow, described above, STCDF incurs an error bounded by 1 LSB in chopping mode and by 1/2 LSB in rounding mode.

STCFI/STCFL/STCDI/STCDL

STORE AND CONVERT FROM FLOATING OR DOUBLE
TO INTEGER OR LONG INTEGER

175(AC+4)DST



Format: STCFI AC,DST

Operation: (DST) <-- Cxj(AC) if -JL - 1 < Cxj(AC) < JL + 1, else (DST) <-- 0, where Cjx specifies conversion from floating mode x to integer mode j.

```
j = I if FL = 0, j = L if FL = 1
x = F if FD = 0, x = D if FD = 1
```

JL is the largest integer.

```
2 ** 15 - 1 for FL = 0
2 ** 32 - 1 for FL = 1
```

Condition Codes:

```
C, FC <-- 0 if -JL - 1 < Cxj(AC) < JL + 1, else
C, FC <-- 1
V, FV <-- 0
Z, FZ <-- 1 if (DST) = 0, else Z, FZ <-- 0
N, FN <-- 1 if (DST) < 0, else N, FN <-- 0
```

Description: Conversion is performed from a floating-point representation of the data in the accumulator to an integer representation.

If the conversion is to a 32-bit word (L mode), and an addressing mode of 0 or immediate addressing mode is specified, only the most significant 16 bits are stored in the destination register.

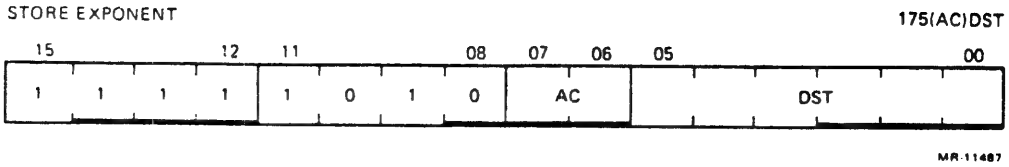
If the operation is out of the integer range selected by FL, FC is set to 1 and the contents of the DST are set to 0.

Numbers to be converted are always chopped (rather than rounded) before they are converted. This is true even when the chop mode bit FT is cleared in the FPS register.

Interrupts: These instructions do not interrupt if FIUV is enabled, because the -0, if present, is in AC, not in memory. If FIC is enabled, trap on conversion failure will occur.

Accuracy: These instructions store the integer part of the floating-point operand, which may not be the integer most closely approximating the operand. They are exact if the integer part is within the range implied by FL.

STEXP



Format: STEXP AC,DST

Operation: (DST) <-- EXP(AC) - 200

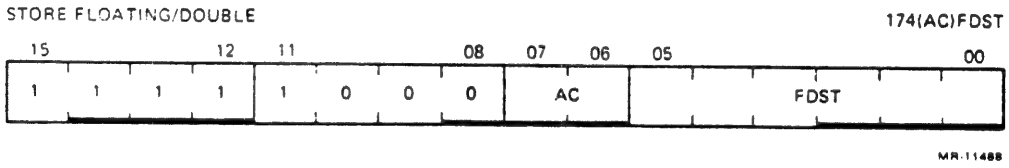
Condition Codes:
C, FC <-- 0
V, FV <-- 0
Z, FZ <-- 1 if (DST) = 0, else Z, FZ <-- 0
N, FN <-- 1 if (DST) < 0, else N, FN <-- 0

Description: Convert AC's exponent from excess 200 notation to 2's complement and store the result in DST.

Interrupts: This instruction will not trap on -0. Overflow and underflow cannot occur.

Accuracy: This instruction is exact.

STF/STD



Format: STF AC,FDST

Operation: (FDST) <-- AC

Condition Codes:
FC <-- FC
FV <-- FV
FZ <-- FZ
FN <-- FN

Description: Store single- or double-precision number from AC.

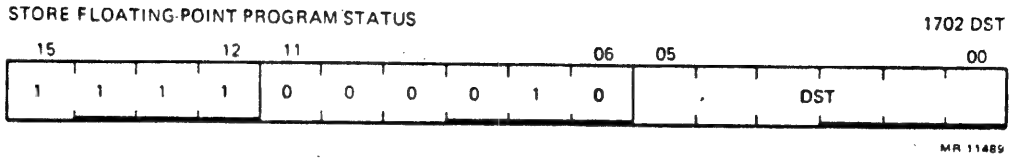
Interrupts: These instructions do not interrupt if FIUV is enabled, because the -0, if present, is in AC, not in memory. Overflow and underflow cannot occur.

Accuracy: These instructions are exact.

Special Comment: These instructions permit storage of a -0 in

memory from AC. There are two conditions in which -0 can be stored in an AC of the DCJ11. One occurs when underflow or overflow is present and the corresponding interrupt is enabled. A second occurs when an LDF or LDD instruction is executed and the FIUV bit is disabled.

STFPS



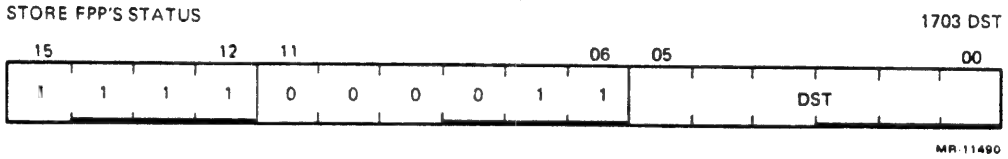
Format: STFPS DST

Operation: (DST) <-- FPS

Description: Store the floating-point status register in DST.

Special Comment: Bits 13, 12, and 4 are loaded with 0. All other bits are the corresponding bits in the FPS.

STST



Format: STST DST

Operation: (DST) <-- FEC
(DST + 2) <-- FEA

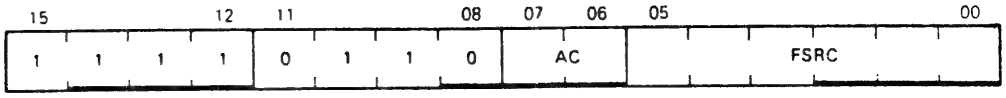
Description: Store the FEC and FEA in DST and DST+2. Note the following.

1. If the destination mode specifies a general register or immediate addressing, only the FEC is saved.

2. The information in these registers is current only if the most recently executed floating-point instruction caused a floating-point exception.

SUBF/SUBD

SUBTRACT FLOATING/DOUBLE 173(AC)FSRC



MR-11491

Format: SUBF FSRC,AC

Operation: Let DIFF = (AC) - (FSRC)

If underflow occurs and FIU is not enabled, AC <-- exact 0.

If overflow occurs and FIV is not enabled, AC <-- exact 0.

For all others cases, AC <-- DIFF.

Condition Codes:
FC <-- 0
FV <-- 1 if overflow occurs, else FV <-- 0
FZ <-- 1 if (AC) = 0, else FZ <-- 0
FN <-- 1 if (AC) < 0, else FN <-- 0

Description: Subtract the contents of FSRC from the contents of AC. The subtraction is carried out in single- or double-precision and is rounded or chopped in accordance with the values of the FD and FT bits in the FPS register. The result is stored in AC except for:

1. Overflow with interrupt disabled.
2. Underflow with interrupt disabled.

For these exceptional cases, an exact 0 is stored in AC.

Interrupts: If FIUV is enabled, trap on -0 in FSRC occurs before execution. If overflow or underflow occurs, and if the corresponding interrupt is enabled, the trap occurs with the faulty result in AC. The fractional parts are correctly stored. The exponent part is too small by 400 for overflow. It is too large by 400 for underflow, except for the special case of 0, which is correct.

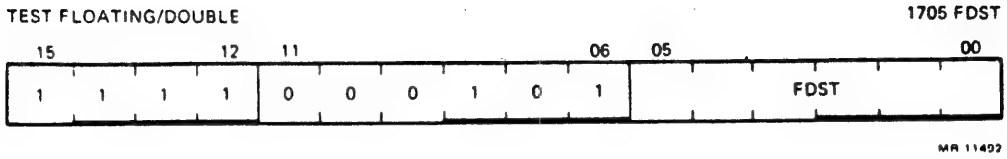
Accuracy: Errors due to overflow and underflow are described above. If neither occurs: for like-signed operands with exponent difference of 0 or 1, the answer returned is exact if a loss of significance of one or more bits can occur. Note that these are the only cases for which loss of significance of more than one bit can occur. For all other cases the result is inexact with error bounds of:

1. LSB in chopping mode with either single- or double-precision.

2. 1/2 LSB in rounding mode with either single- or double-precision.

Special Comment: The undefined variable -0 can occur only in conjunction with overflow or underflow. It will be stored in AC only if the corresponding interrupt is enabled.

TSTF/TSTD



Format: TSTF FDST

Operation: (FDST)

Condition Codes:
FC <-- 0
FV <-- 0
FZ <-- 1 if (FDST) = 0, else FZ <-- 0
FN <-- 1 if (FDST) < 0, else FN <-- 0

Description: Set the floating-point condition codes according to the contents of FDST.

Interrupts: If FIUV is set, trap on -0 occurs before execution. Overflow and underflow cannot occur.

Accuracy: These instructions are exact.

8.1 INTRODUCTION

This chapter covers topics related to the interfacing of external logic to the DCJ11.

8.2 GENERAL-PURPOSE (GP) CODES

An important means of communicating with external logic is through the use of GP Reads and Writes (see Chapter 3 - Bus Cycles). GP Reads and Writes are associated with codes that specify the function performed during the GP Read or Write cycle. External logic interprets these codes to implement system functions. Table 8-1 summarizes the GP codes.

Table 8-1 GP Codes and Functions

GP Code (octal)	GP Read or Write	Function
000	Read	Reads the power-up mode, HALT option, FPA option, POK, and boot address.
001	Read	Reads FPA data (if FPA exists)
002	Read	Reads the power-up mode, HALT option, FPA option, POK, and boot address, and (if an FPA exists) clears the FPA's FPS.
003	Read	Acknowledges FPE and reads the FEC (floating exception code) register
003	Write	Writes FPA 16-bit data (if FPA exists)
014	Write	Asserts bus reset signal
034	Write	Signals exit from console ODT
040	Write	Reserved for future use
100	Write	Acknowledges EVENT
140	Write	Acknowledges power fail
214	Write	Negates bus reset signal
220	Write	Microdiagnostic test 1 passed
224	Write	Microdiagnostic test 2 passed
230	Write	Microdiagnostic test 3 passed
234	Write	Signals entry into console ODT

Specific external logic designs may need to interpret only a subset of the GP codes. For example, a minimal system with no FPA and no need for POK or a bus reset signal would only have to identify a GP code associated with the reading of power-up configuration data during the DCJ11's initializtion sequence. As shown in the flowchart in Paragraph 8.3.2, this is GP code 002.

8.3 POWER-UP AND INITIALIZATION

The DCJ11 performs a specific sequence of events at power-up or when it is initialized. These initialization microroutines are described in this paragraph. Also, during power-up the DCJ11 reads the contents of a configuration register to determine its initial mode of operation. This configuration register is also described. A typical power-up circuit is also provided.

8.3.1 Initialization Timing - Initialization timing is shown in Figure 8-1. When external logic asserts INIT for a minimum of 25 clock periods, the DCJ11 is forced into a power-up initialization sequence. As shown in Figure 8-1, the DCJ11 asserts $\overline{SCTL}$ shortly after the assertion of $\overline{INIT}$. $\overline{SCTL}$ is deasserted approximately five clock periods after $\overline{INIT}$ is deasserted.

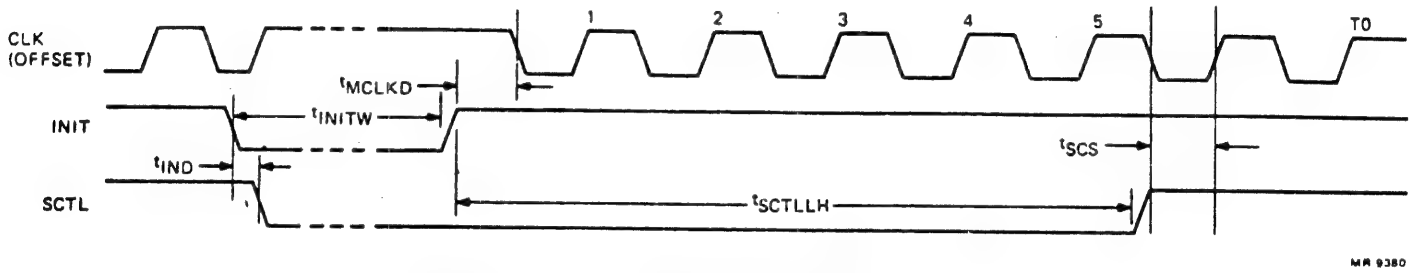


Figure 8-1 Initialization

8.3.2 Initialization Microroutine - The microroutine that is executed when the DCJ11 is powered up or initialized is shown in Figure 8-2. Note that GP codes that indicate some event (such as the passing of a microdiagnostic test) can be used by external logic to light LEDs for a visual indication of the event.

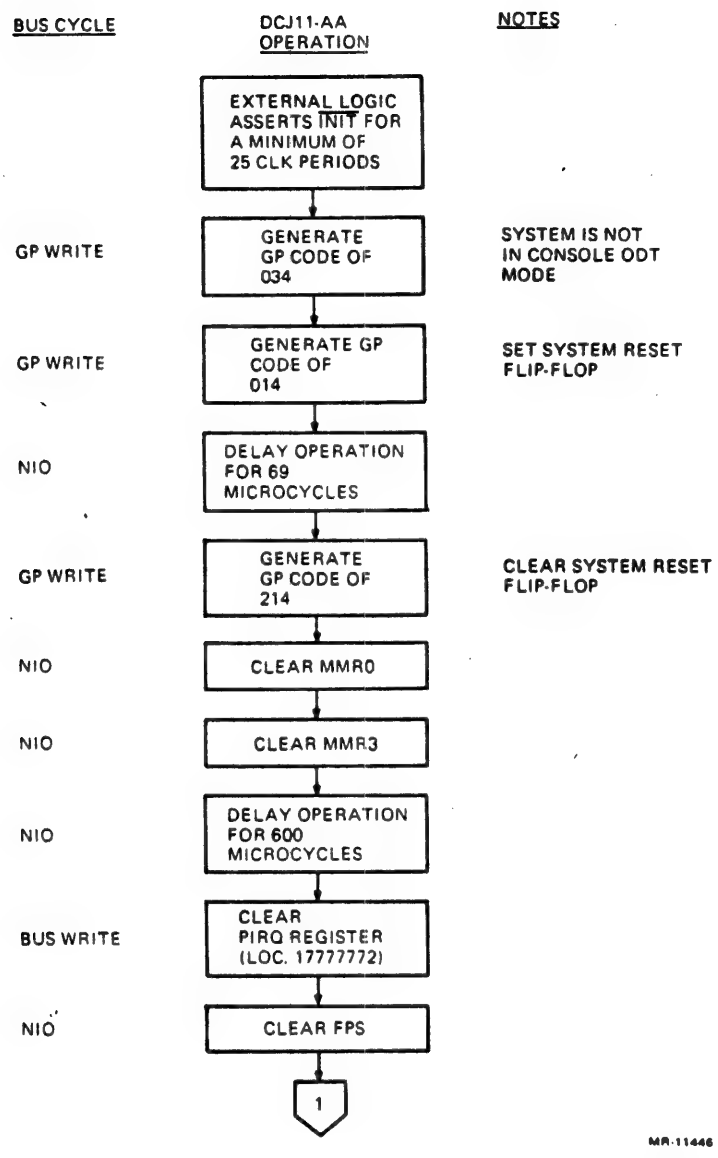


Figure 8-2 Initialization Sequence

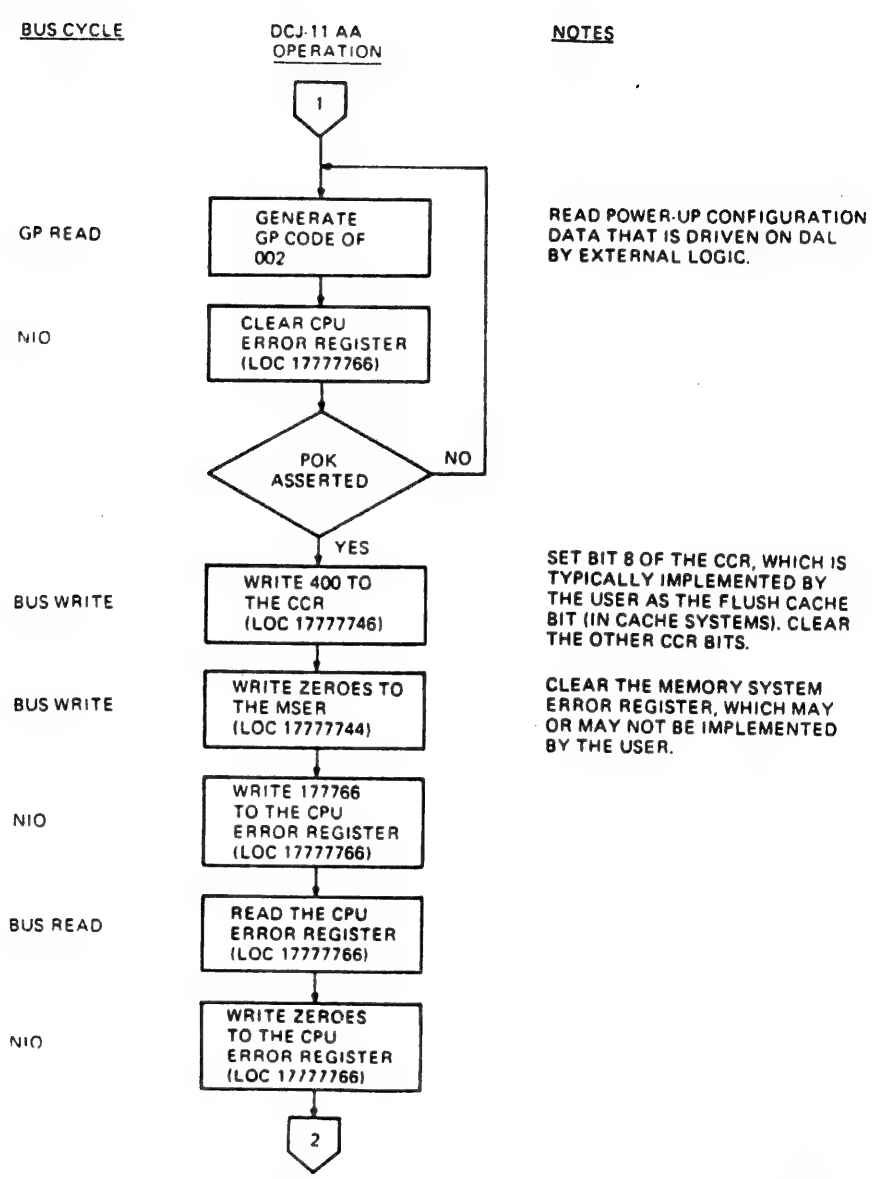


Figure 8-2 Initialization Sequence (Continued)

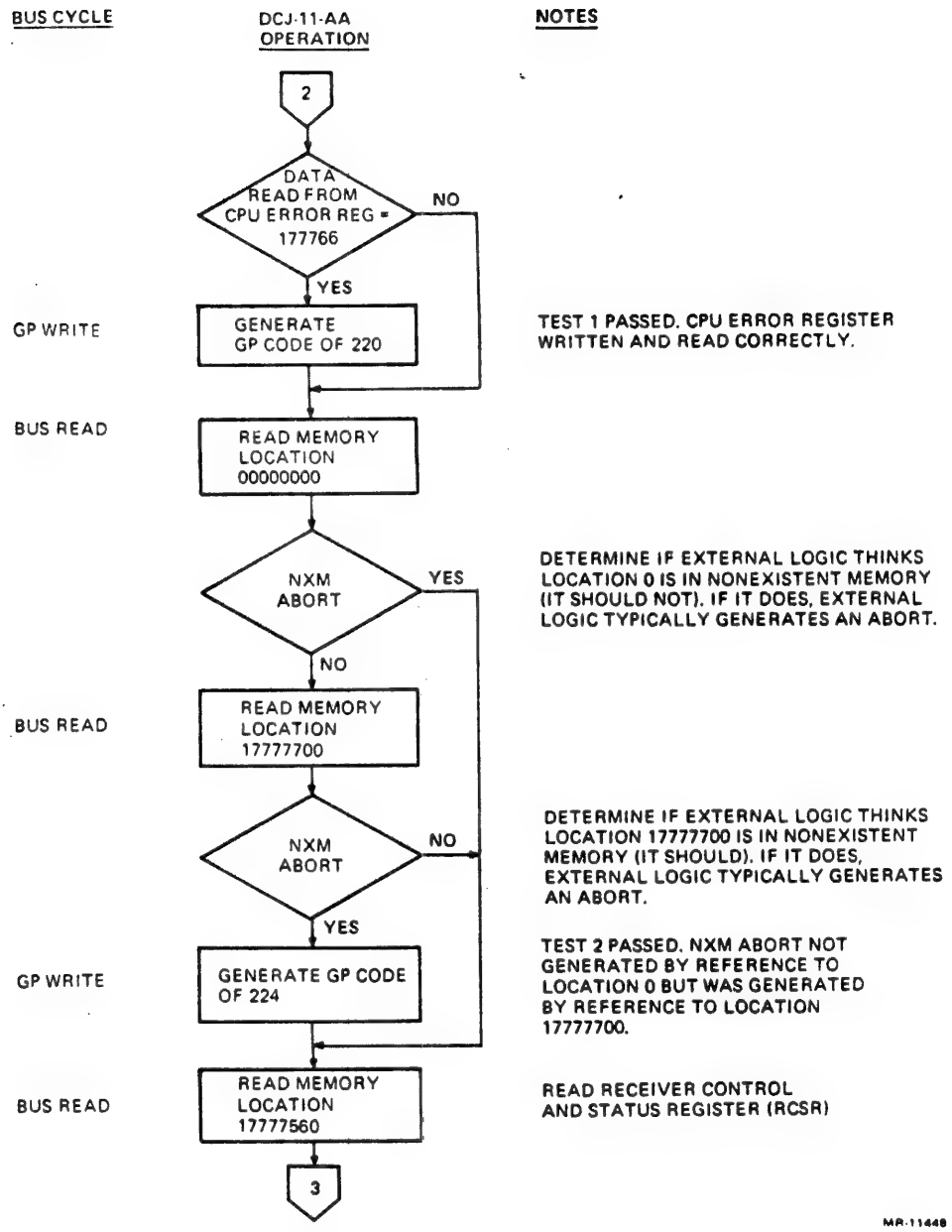


Figure 8-2 Initialization Sequence (Continued)

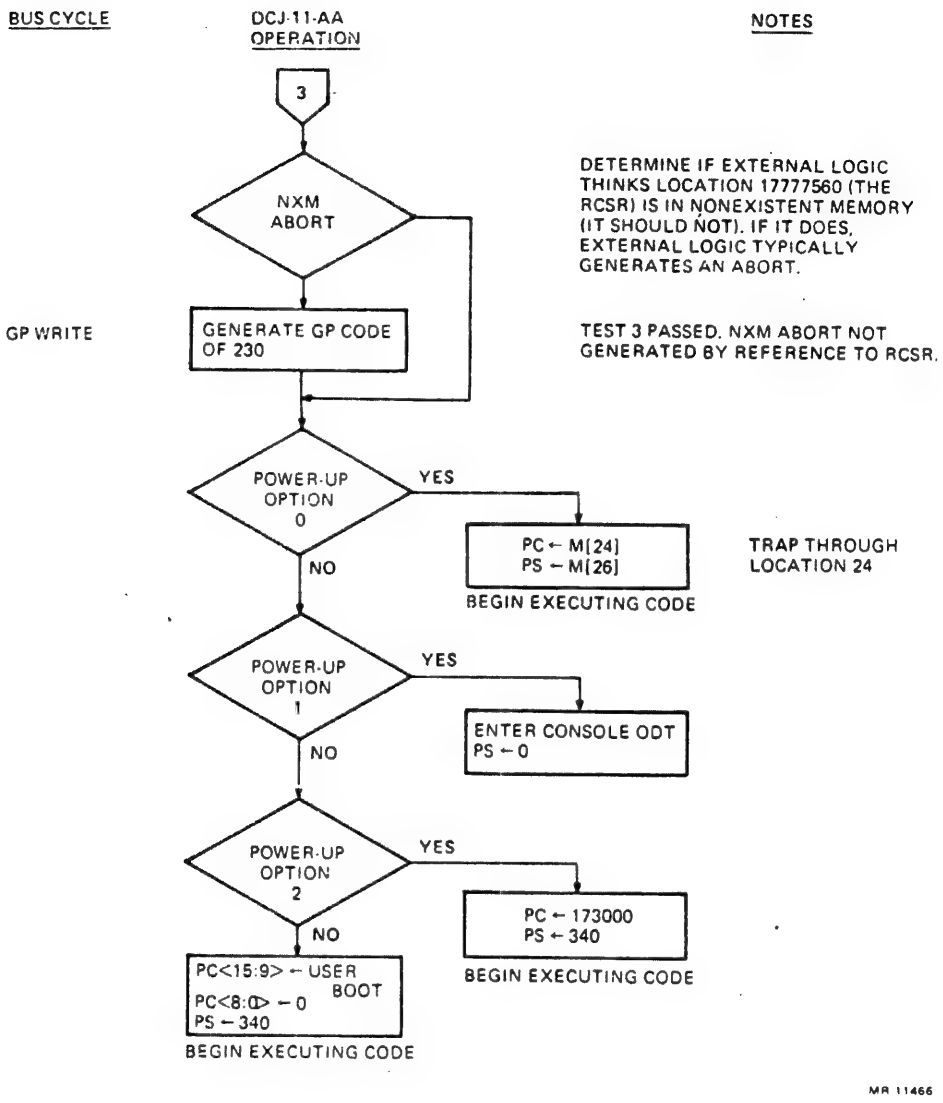


Figure 8-2 Initialization Sequence (Continued)

8.3.3 Power-Up Configuration - The power-up configuration is specified by setting bits in an external register which is read (via the DAL) during the DCJ11's initialization sequence. It specifies various user-defined initial conditions. The register is shown in Figure 8-3.

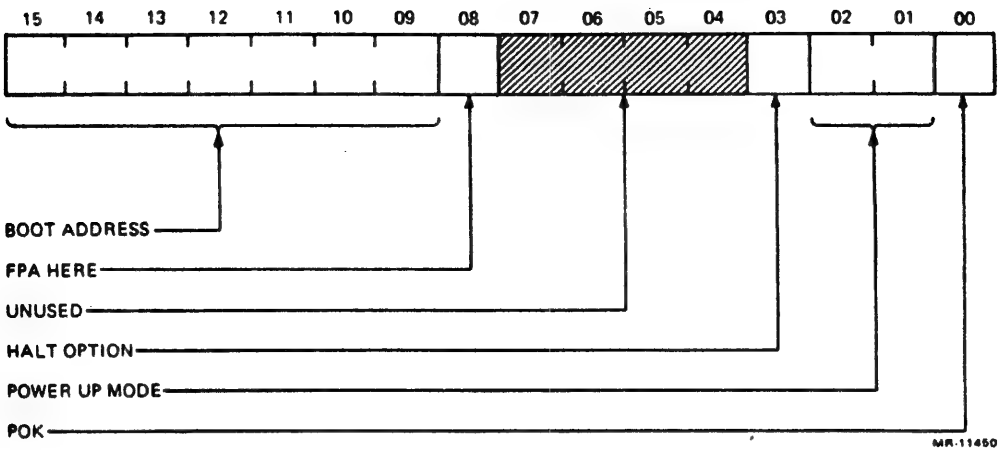


Figure 8-3 Power-Up Configuraton Register

Bit(s)	Name	Description
<15:9>	Boot Address	Contains the most significant seven bits (bits <15:9>) of a user-defined boot address used in power-up mode 3. The lower bits of the boot address (bits <8:0>) are zeroes.
8	FPA Here	Indicates the presence of an optional floating-point accelerator (FPA) when set. When cleared, the FPA is not present.
<7:4>	Unused	These bits are not interpreted by the DCJ11.
3	Halt Option	Indicates how a HALT instruction will execute in kernel mode. If set, the DCJ11 traps through location 4 and sets bit 7 of the CPU error register when HALT is executed. If cleared, the DCJ11 enters console ODT when HALT is executed.
<2:1>	Power-Up Mode	Indicates one of four power-up mode options.

Bits 2	Bits 1	Mode
0	0	Trap through location 24
0	1	Enter console ODT
1	0	Power-up to 17773000
1	1	Power-up to the user-defined address specified by bits <15:9>

0	POK	Indicates whether the power supply

is operating within its normal range. Set when power is at an acceptable value.

8.3.4 Power-Up Circuit - A circuit such as that shown in Figure 8-4 can be used to power-up the DCJ11.

INIT is provided to the DCJ11 by power-up logic and the AIO code is latched by the assertion of ALE. The decoder indicates whether a GP Read of 000 or 002 is being executed.

In this simple application, only DAL<8,3:0> are affected by the power-up configuration register. The register is configured to indicate that no FPA is present, power-up mode 0 (trap through location 24) is selected, and power is always OK. The DAL is driven with configuration data when BUFCTL is asserted and a GP Read with a code of 000 or 002 occurs.

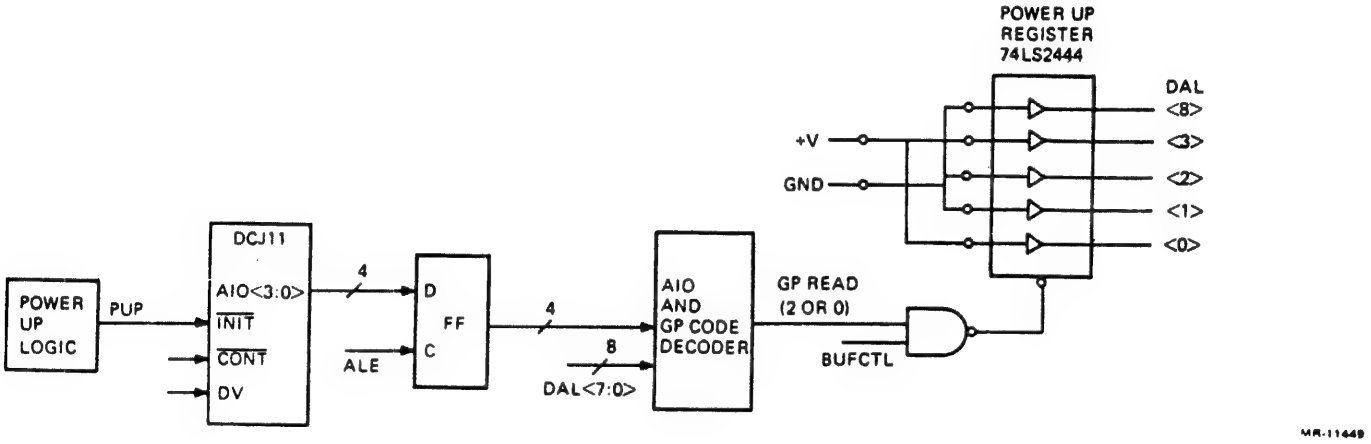


Figure 8-4 Power-Up Circuit

8.4 OTHER MICROROUTINES

Figures 8-5 and 8-6 illustrate two other microroutines whose operation can be monitored by external logic: the power-down microroutine and the console ODT response to entering the "go" command.

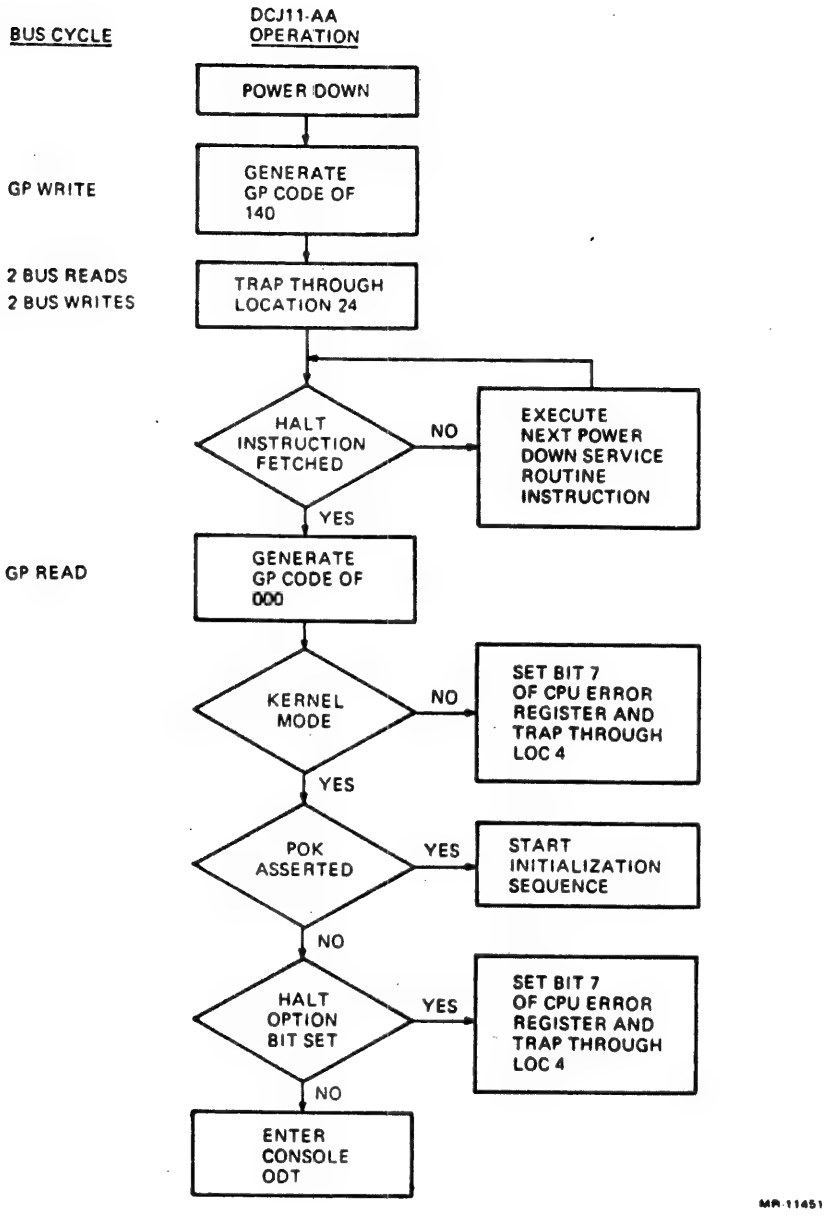


Figure 8-5 Power-Down Sequence

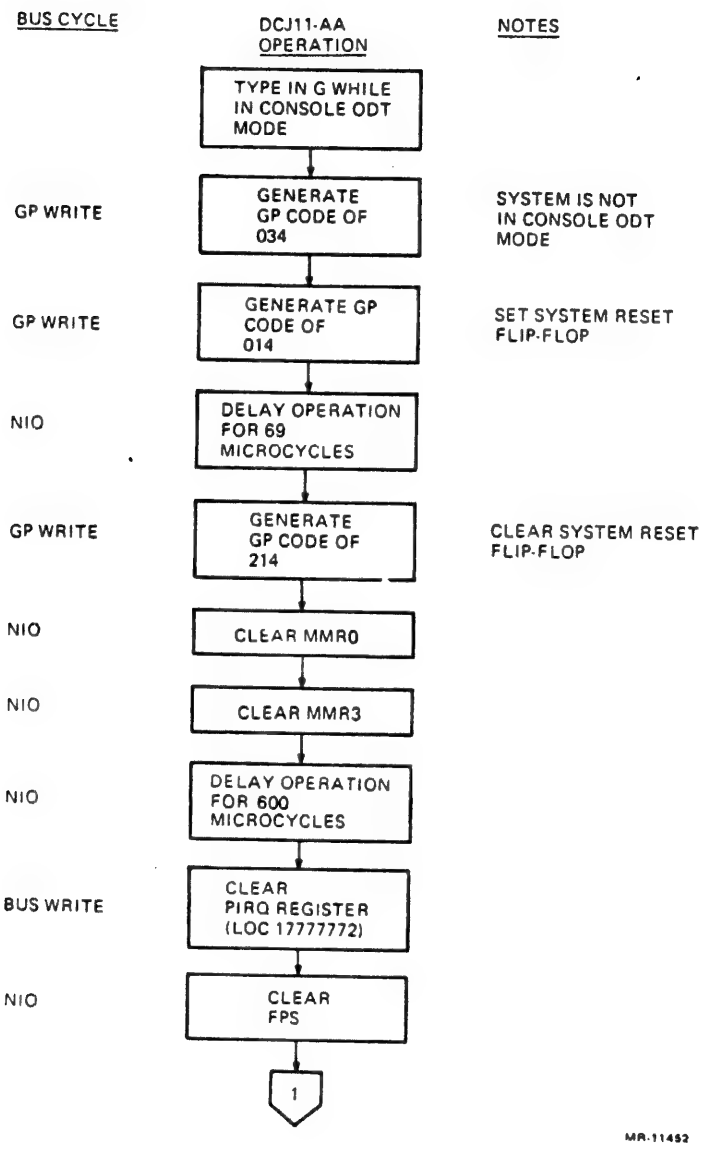


Figure 8-6 Console Start Sequence

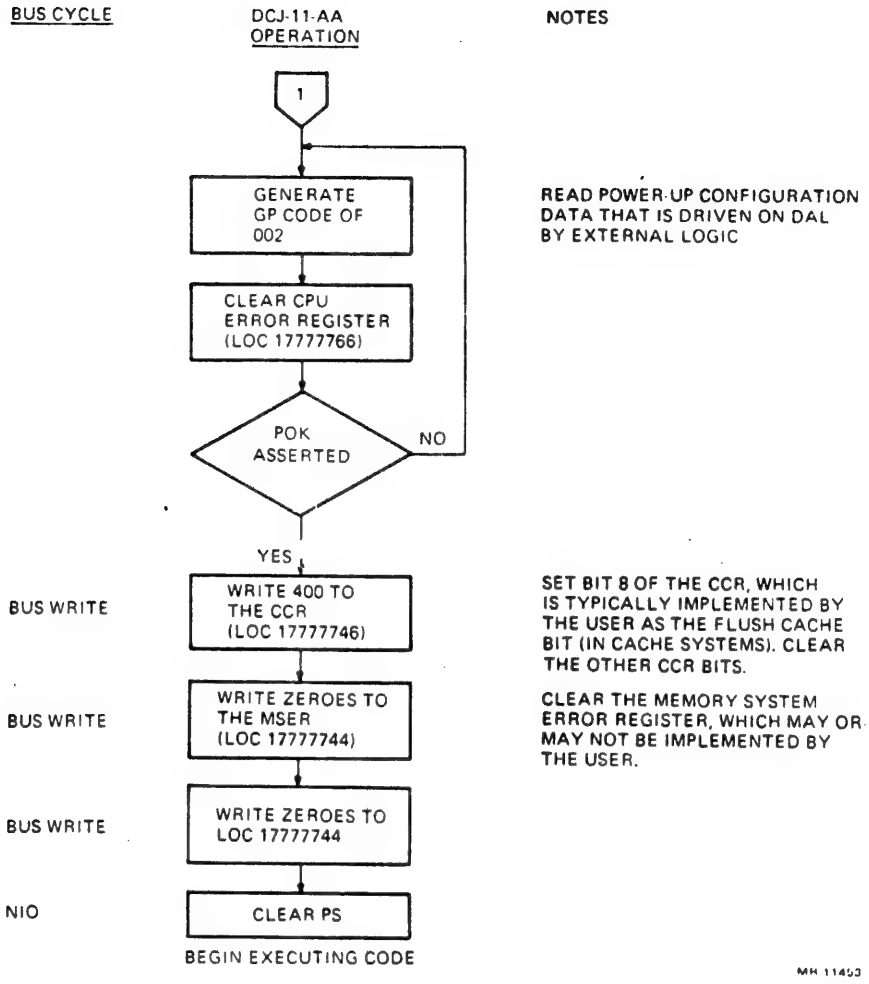


Figure 8-6 Console Start Sequence (Continued)

Absolute Maximum Rating

Storage Temperature Range:	-65 C to +150 C
Active Temperature Range:	-55 C to +125 C
Supply Voltage:	+7.0V
Input or Output Voltage Applied:	Vss -0.3V Vcc +0.3V

Electrical Characteristics

Specified Temperature Range	0 C to +70 C
Specified Voltage Range	+4.75V to +5.25V
Test Conditions	Temperature = +70 C Vss = 0V Vcc = +4.75V (except as noted)

Symbol	Parameter	Min.	Max.	Units	Test Condition
V_{IH}	High level MOS input	70% V_{cc}		V	
V_{IL}	Low level MOS input		30% V_{cc}	V	
V_{IHT}	High level TTL input	2.2		V	
V_{ILT}	Low level TTL input		0.8	V	
I_I	Input leakage current except TEST inputs (note 1)	-10.0	10.0	uA	$0V \leq V_I \leq V_{cc}$
I_{ILL}	Input current TEST inputs (note 1)	0.1	5.0	mA	$V_I = 0V$
I_{OH}	Output current at high level	-2.0		mA	$V_O = V_{cc} - 0.4V$
I_{OL}	Output current at low level	2.0		mA	$V_O = 0.4V$

Symbol	Parameter	Min.	Max.	Units	Test Condition
I_{OHT}	Output current at high TTL level	-2.0		mA	V_O = 2.4V
I_{OSH}	High level sustainer current (note 1)	-0.2	-0.6	mA	$V_O = V_{cc} - 1.0V$
I_{OSL}	Low level sustainer current (note 1)	0.2	0.6	mA	V_O = 1.0V
I_{OZ}	Output leakage current (notes 1,2)	-10.0	10.0	uA	$0V \leq V_O \leq V_{cc}$
I_{CCSB}	Static power supply current (notes 1,3)		30	mA	
C_{IN}	Input capacitance (note 4)		7	pF	
C_{IO}	Input/output capacitance (note 4)		15	pF	
C_{OUT}	Output capacitance (note 4)		15	pF	
C_{MAX}	DCJ11 capacitance plus external capacitance		50	pF	

NOTES

1. Tested at Vcc = 5.25V.
2. Only applies in the high impedance condition.
3. With TEST1 and TEST2 asserted, all outputs open circuit, and all other inputs equal to Vcc.
4. Sampled and guaranteed, but not tested. Does not apply to TEST1 or TEST2.

SIGNAL SUMMARY

TYPE	NAME	APPLICABLE DC TEST
TTL INPUT	IRQ<3:0>, HALT, PWRF, EVENT, PARITY, DV, MISS, CONT, DMR, INIT, FPE	V_{IHT}, V_{ILT}, I_I
TTL OUTPUT	DAL<21:16>, AIO<3:0>, ALE, BUFCTL, SCTL, STRB, BS<1:0>, MAP, PRDC	I_{OL}, I_{OHT}, I_{OZ}
MOS INPUT	TEST1, TEST2	V_{IH}, V_{IL}, I_{ILL}
MOS OUTPUT	CLK, CLK2	I_{OH}, I_{OL}, I_{OZ}
TTL I/O	ABORT*	V_{ILT}, I_{OL}, I_{OHT}, I_{OZ}, I_{OSH}
TTL I/O	DAL<15:00>	V_{IHT}, V_{ILT}, I_{OL}, I_{OHT}, I_{OZ}
Power	Vcc	I_{CCSB}

* ABORT must be driven with an open collector driver because the DCJ11 has a pull-up device that supplies I_{OSH}.

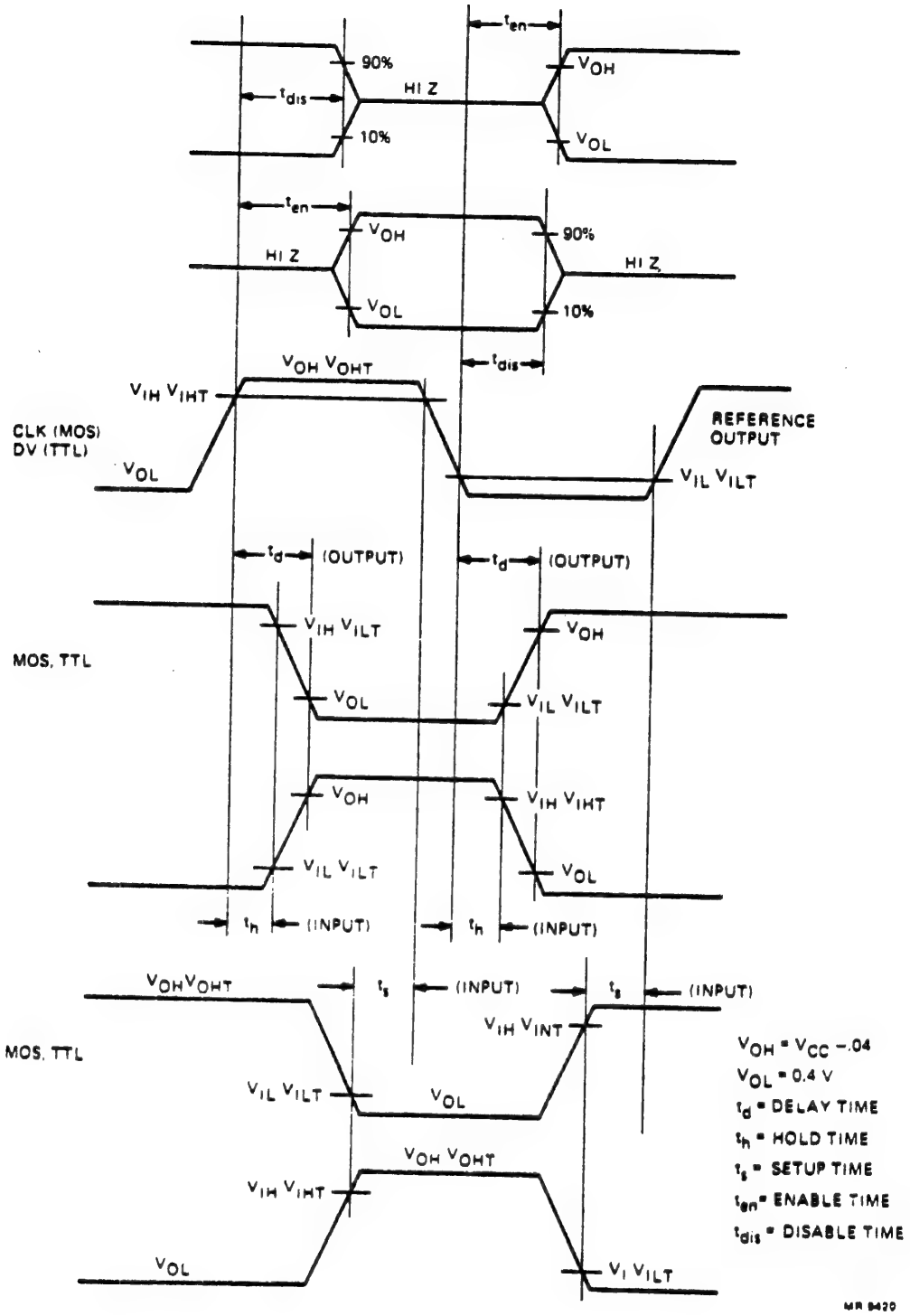


Figure A-1 Voltage Waveforms

Test Conditions:

Temperature = +70 C
Vss = 0V
Vcc = +4.75V (except as noted)
CMAX = 50 pF

Timing Requirements

Symbol	Parameter	Min	Max	Units
t_{INITW}	$\overline{INIT}$ pulse width	10		clock periods
t_{SCTLLH}	Initialization interval	225		ns
t_{DS}	DAL<15:00> setup, with respect to T3	35		ns
t_{DH}	DAL<15:00> hold, with respect to T3	20		ns
t_{DVDS}	DAL<15:00> setup, with respect to DV	35		ns
t_{DVDH}	DAL<15:00> hold, with respect to DV	35		ns
t_{DVW}	DV Pulse width	35		ns
t_{DVF}	DV Fall time		15	ns
t DVH	DV deassertion with respect to T6.	0		ns
t DVS	DV deassertion with respect to T4.	0		ns
t_{HMS}	$\overline{MISS}$ setup	30		ns
t_{HMH}	$\overline{MISS}$ hold	10		ns
t_{SVCS}	IRQ<3:0>, $\overline{HALT}$, $\overline{PWRF}$, $\overline{FPE}$, $\overline{EVENT}$ setup (see note)	20		ns

Symbol	Parameter	Min	Max	Units
t_{SVCH}	IRQ<3:0>, HALT, PWRF, FPE, EVENT hold (see note)	20		ns
t_{PARS}	PARITY setup (see note)	20		ns
t_{PARH}	PARITY hold (see note)	20		ns
t_{ABS}	ABORT drive	30		ns
t_{ABD}	ABORT delay	0		ns
t_{ABW}	ABORT width	40 + t_{CLKH}		ns
t_{CNTS}	CONT setup (see note)	30		ns
t_{CNTH}	CONT hold (see note)	20		ns
t_{DMRS}	DMR setup (see note)	30		ns
t_{DMRH}	DMR hold (see note)	20		ns

Note:
Setup and hold requirements are only to guarantee recognition at next sample point.

Timing Responses

Symbol	Parameter	Min	Max	Units	Figure References
t_{CYCLE}	CLK cycle time	67		ns	B-1, B-4
t_{CLKH}	CLK high width	28		ns	B-1, B-4
t_{CLKL}	CLK low width	28		ns	B-1, B-4
t_{R}	CLK rise time		7	ns	B-1, B-4
t_{F}	CLK fall time		7	ns	B-1, B-4
t_{PCYC}	CLK2 cycle time	67		ns	B-1, B-3
t_{PCLKH}	CLK2 high width	28		ns	B-1, B-3

Symbol	Parameter	Min	Max	Units	Figure References
t_{PCLKL}	CLK2 low width	28		ns	B-1, B-3
t_{PR}	CLK2 rise time		7	ns	B-1, B-3
t_{PF}	CLK2 fall time		7	ns	B-1, B-3
t_{PCLKD}	CLK2 valid delay		tbs	ns	B-1, B-3
t_{MAPD}	MAP delay		45	ns	B-1, B-3
t_{SD}	Strobe active delay	0		ns	B-3
t_{SID}	Strobe inactive delay	0		ns	B-3
t_{DIS}	DAL output disable		35	ns	B-2
t_{DALD}	DAL valid delay		65	ns	B-3
t_{DALH}	DAL valid hold	0		ns	B-3
t_{PD}	PRDC valid delay		50	ns	B-3
t_{PID}	PRDC invalid delay		50	ns	B-3
t_{AIOD}	AIO<3:0> delay		75	ns	B-3

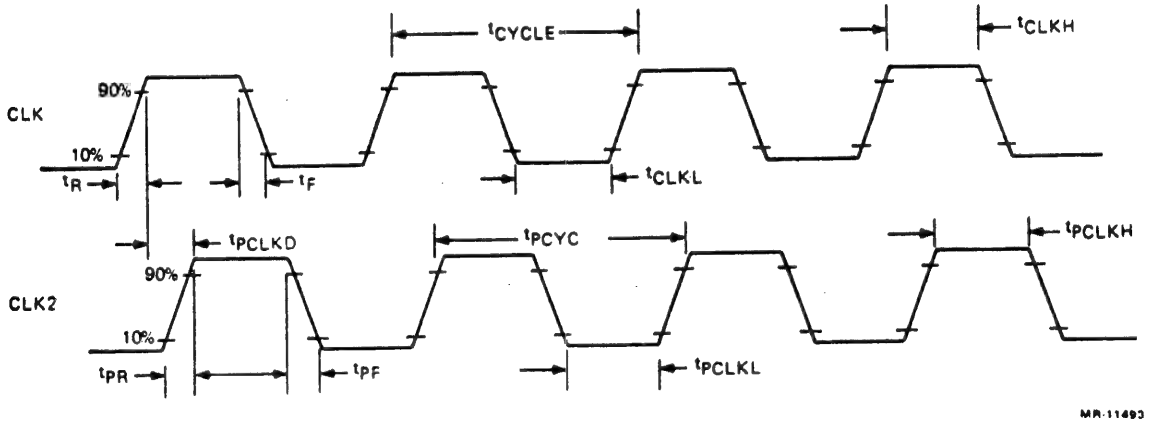


Figure B-1 Clock Timing

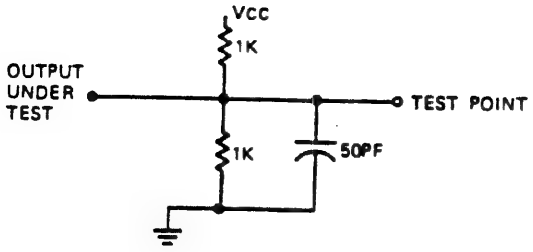


Figure B-2
Three State
Disable Test Circuit

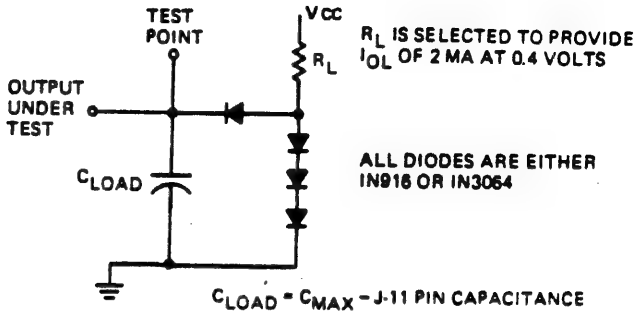


Figure B-3
TTL Output Test Circuit

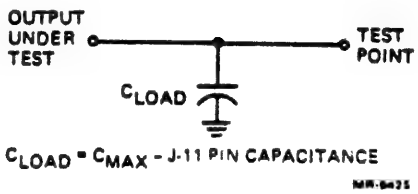


Figure B-4
MOS Output Test Circuit

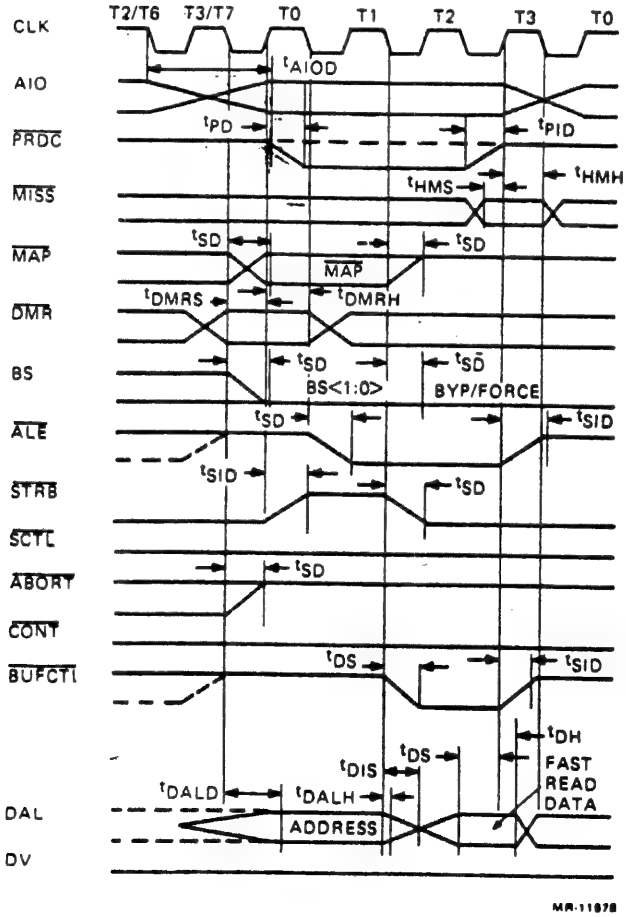


Figure B-5 Non-Stretched Bus Read Timing

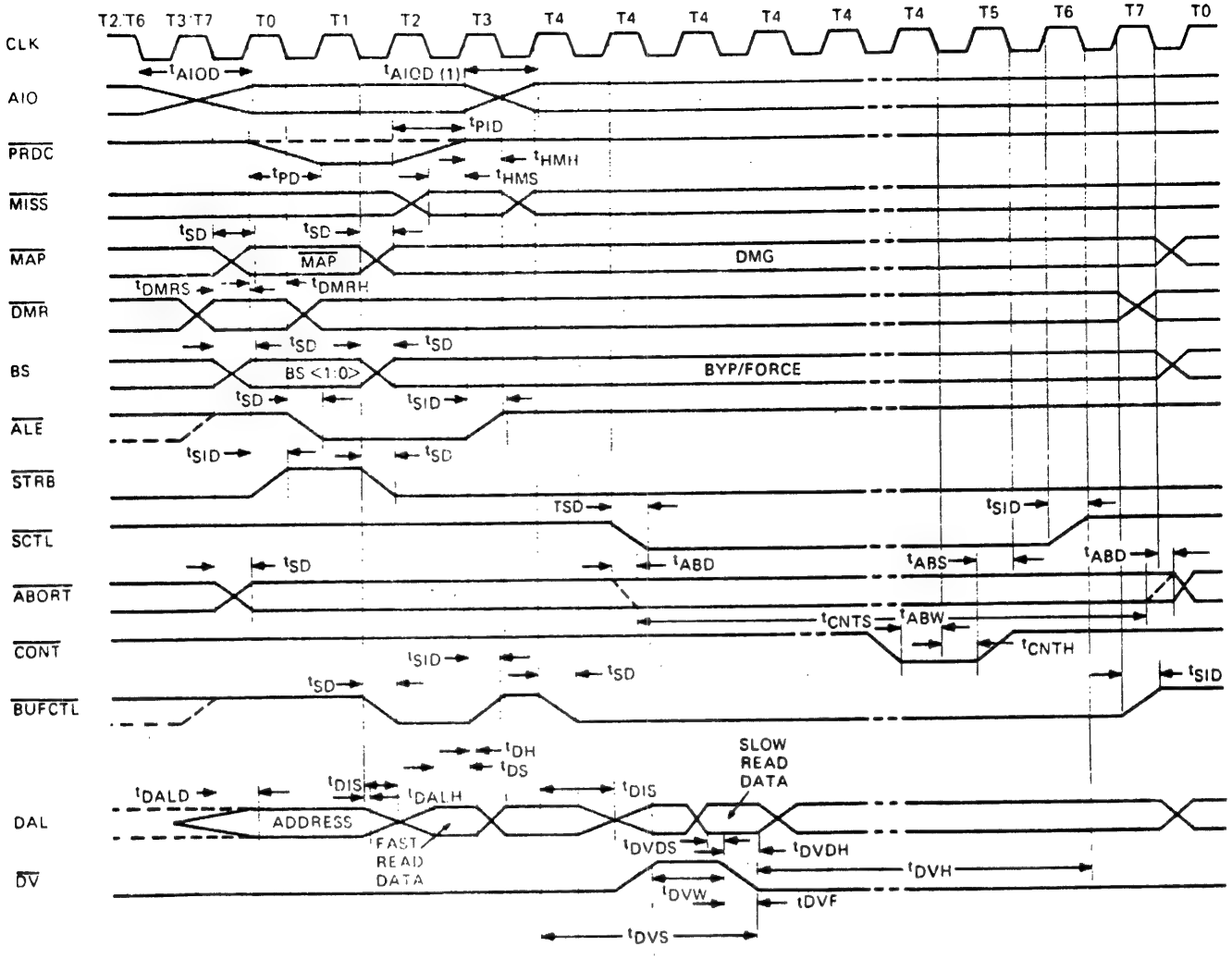


Figure B-6 Stretched Bus Read Timing

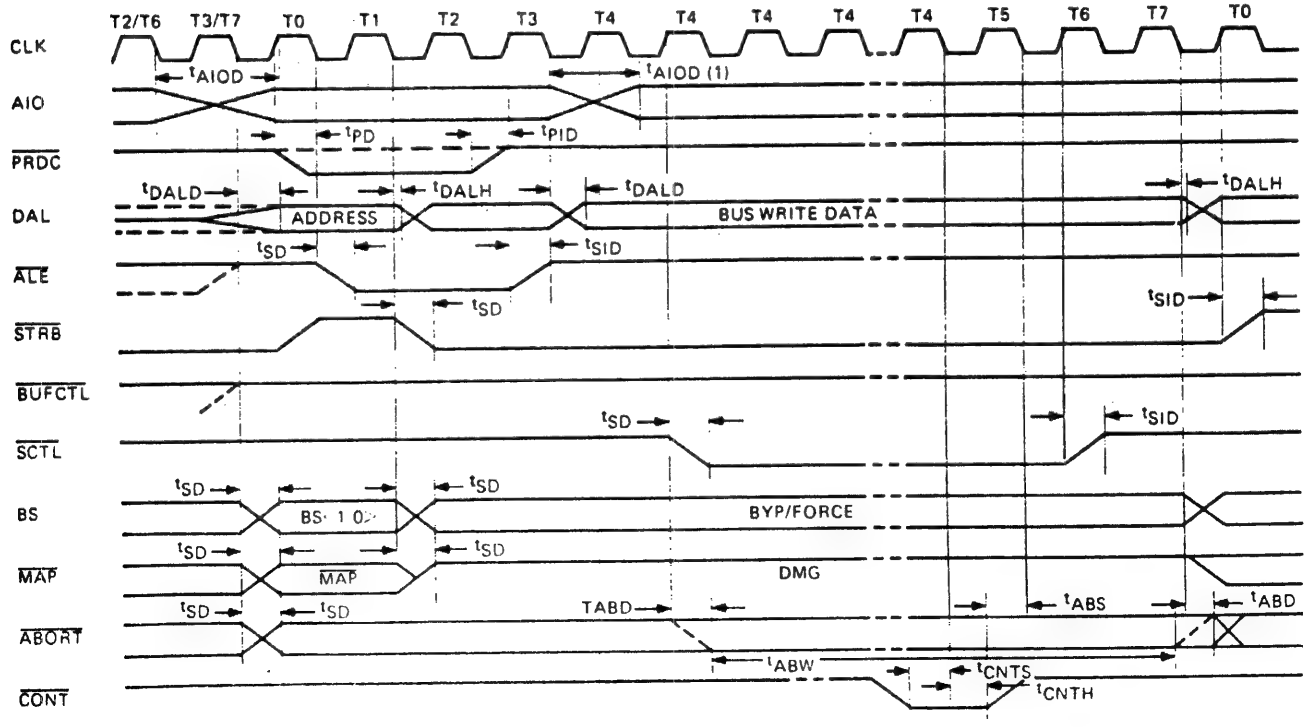


Figure B-7 Bus Write Timing

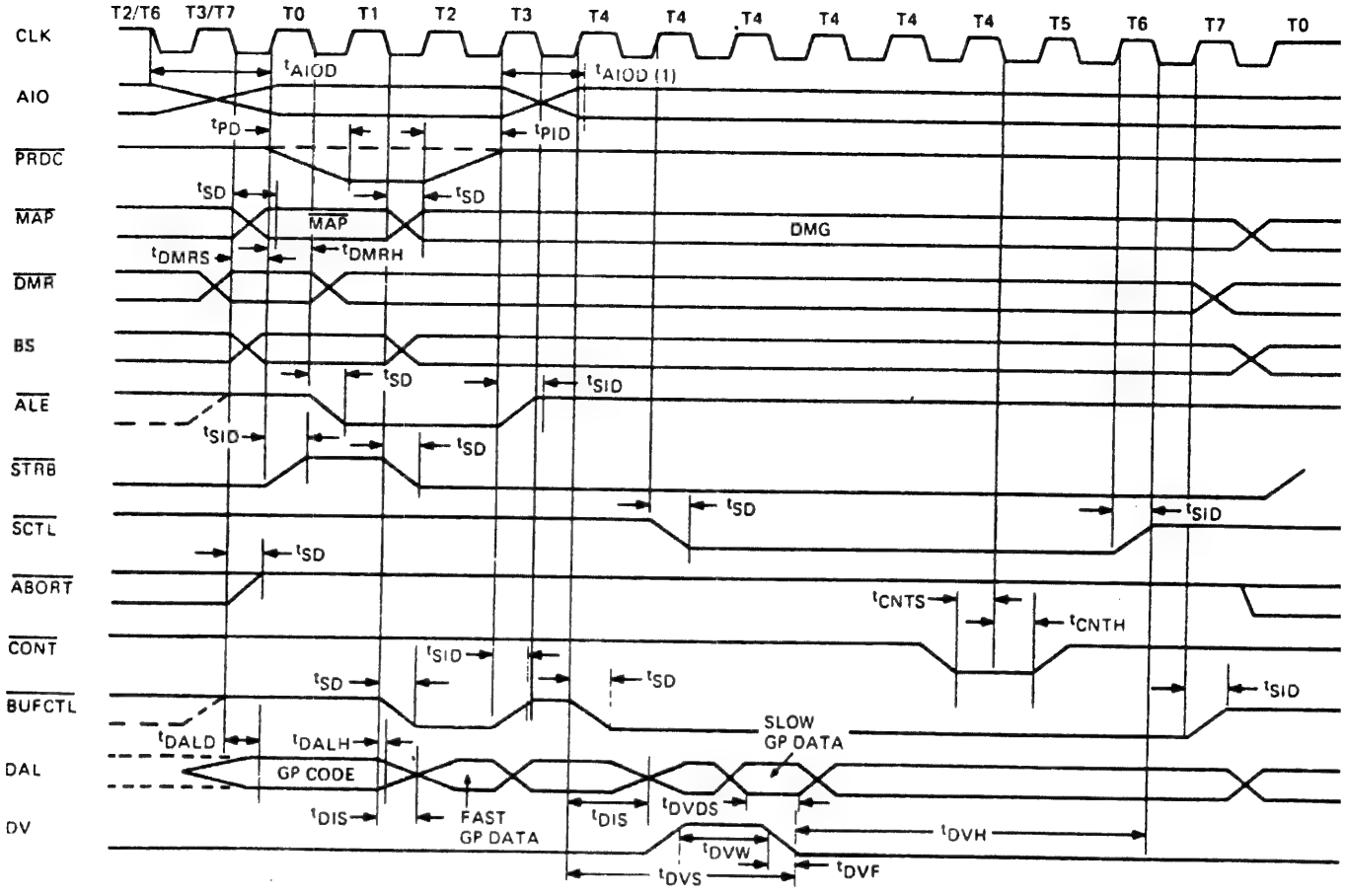


Figure B-8 General-Purpose Read Timing

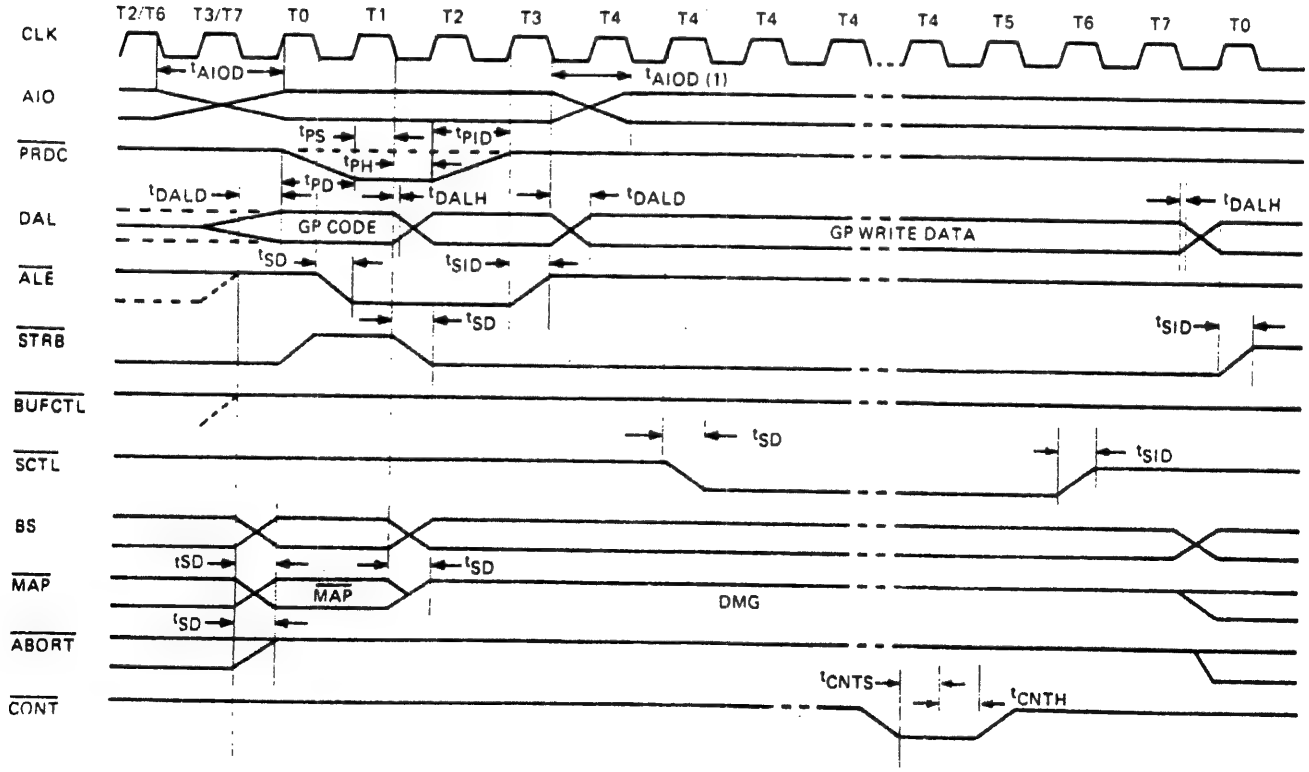


Figure B-9 General-Purpose Write Timing

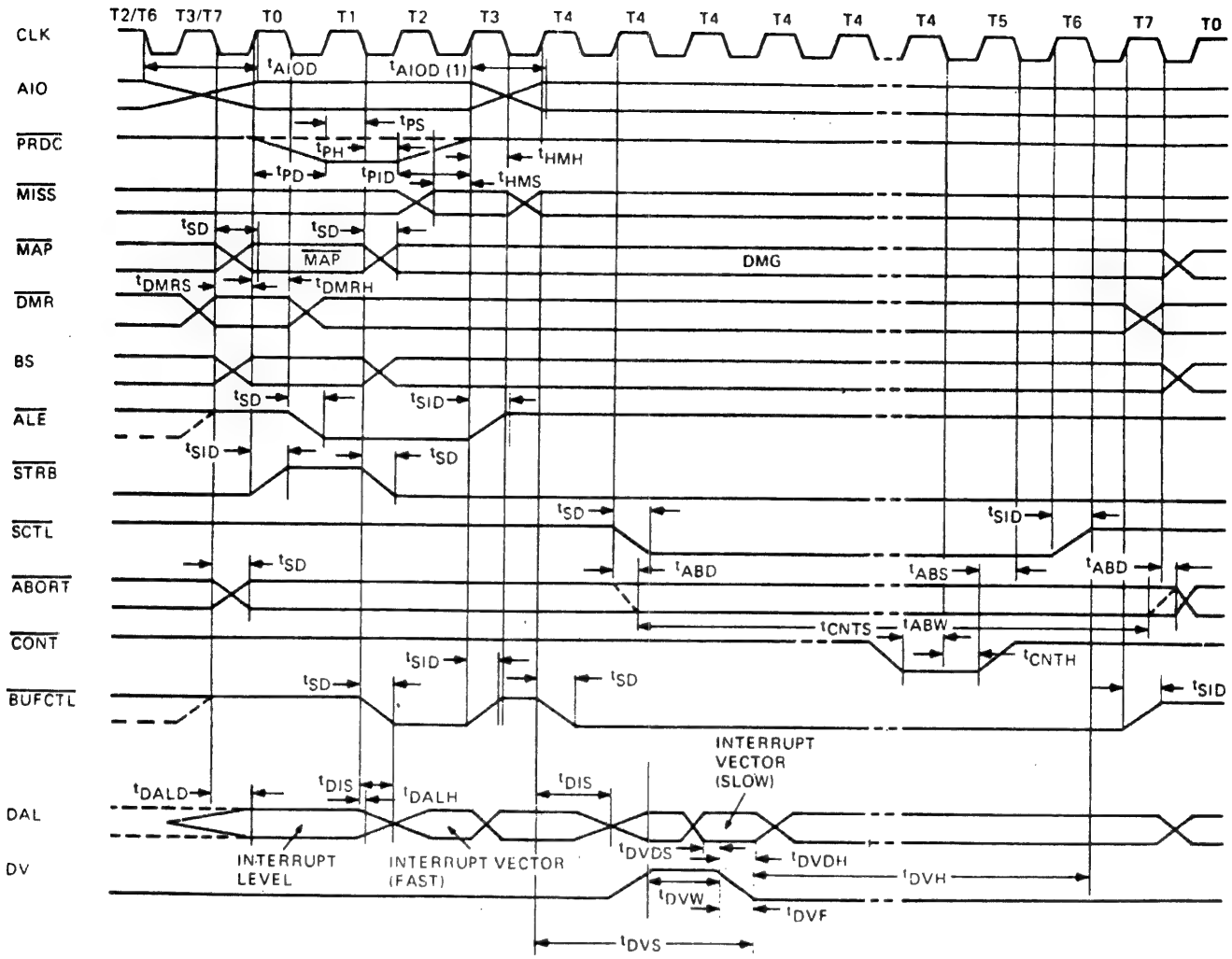


Figure B-10 Interrupt Acknowledge Timing

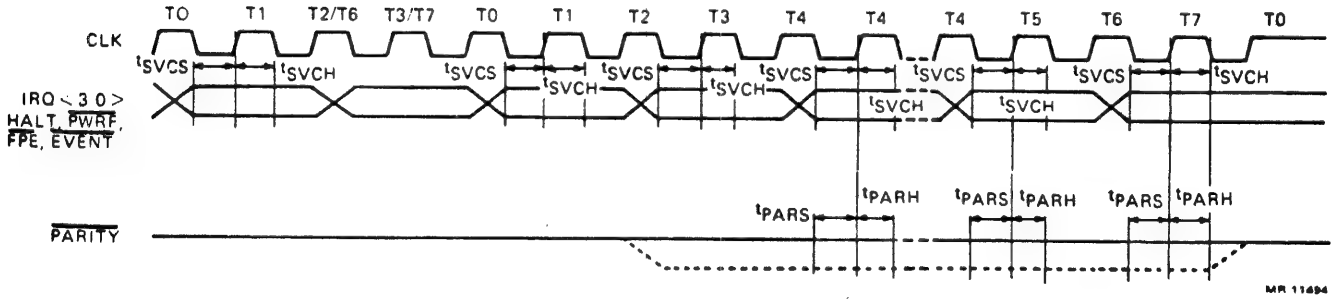


Figure B-11 Interrupt Timing

C.1 HARDWARE DIFFERENCES BETWEEN THE DCJ11 AND THE PDP-11/44

The DCJ11 may replace the PDP-11/44 in certain applications; however, it does not contain the following PDP-11/44 hardware features:

- Cache Data and Maintenance Registers (17777750, 17777754)
- Memory System Error Register (17777744)
- Switch Register (17777570).

The DCJ11 does contain additional functionality not present in the 11/44:

- Dual general register set
- SPL, MTPS, MFPS, TSTSET, WRTLCK instructions.

The following list summarizes the hardware differences between the 11/44 and the DCJ11:

Address	Function	Differences
17777776	PS	Added register set select bit<11>.
17777772	PIRQ	No difference.
17777766	CPU Error	Unibus monitoring bits not implemented.
17777754	Cache Data	Not implemented.
17777752	Hit/Miss	No difference.

17777750	Maintenance	Not implemented.
17777746	Cache Control	Hardware specific changes (see Paragraph 5.2.1).
17777744	Memory Error	Not implemented.
17777676 to 17777660	User Data PAR	No difference.
17777656 to 17777640	User Instruction PAR	No difference.
17777636 to 17777620	User Data PDR	No difference.
17777616 to 17777600	User Instruction PDR	No difference.
17777576	MMR2	No difference.
17777574	MMR1	No difference.
17777572	MMR0	Eliminated maintenance mode.
17777570	Switch Register	Not implemented.
17772516	MMR3	No difference.
17772376 to 17772360	Kernel Data PAR	No difference.
17772356 to 17772340	Kernel Instruction PAR	No difference.
17772336 to 17772320	Kernel Data PDR	No difference.
17772316 to 17772300	Kernel Instruction PDR	No difference.
17772276 to 17772260	Supervisor Data PAR	No difference.

17772256 to 17772240	Supervisor Instruction PAR	No difference.
17772236 to 17772220	Supervisor Data PDR	No difference.
17772216 to 17772200	Supervisor Instruction PDR	No difference.

C.2 HARDWARE DIFFERENCES BETWEEN THE DCJ11 AND THE PDP11/70

The DCJ11 may replace the PDP-11/70 in certain applications; however, it does not contain the following PDP-11/70 hardware features:

- Stack Limit Register (17777774)
- Micro Break Register (17777770)
- System ID Register (17777764)
- System Size Registers (17777760, 17777762)
- Maintenance Register (17777750)
- Memory System Error Register (17777744)
- Physical Error Address Registers (17777740, 17777742)
- Switch Register (17777570).

The DCJ11 does contain additional functionality not present in the 11/70:

- MTPS, MFPS, MFPT, CSM, TSTSET, WRTLCK instructions
- Bypass cache bit in PDRs.

The following list summarizes the hardware differences between the 11/70 and the DCJ11:

Address	Function	Differences
17777776	PS	Added suspended instruction bit <8>.
17777774	Stack Limit	Not implemented.
17777772	PIRQ	No difference.
17777770	Micro Break	Not implemented.
17777766	CPU Error	No difference.
17777764	System ID	Not implemented.
17777762	System Size	Not implemented.
17777760	System Size	Not implemented.
17777752	Hit/Miss	No difference.
17777750	Maintenance	Not implemented.
17777746	Cache Control	Hardware specific changes (see section 5.2.1).
17777744	Memory Error	Not implemented.
17777742	High Error Address	Not implemented.
17777740	Low Error Address	Not implemented.
17777676 to 17777660	User Data PAR	No difference.
17777656 to 17777640	User Instruction PAR	No difference.
17777636 to 17777620	User Data PDR	Added bypass cache, eliminated access flags and access modes other than 0, 2, and 6.
17777616 to 17777600	User Instruction PDR	Added bypass cache, eliminated access flags and access modes other than 0, 2, and 6.
17777576	MMR2	No difference.
17777574	MMR1	No difference.

17777572	MMR0	Eliminated traps, maintenance mode, and instruction complete.
17777570	Switch Register	Not implemented.
17772516	MMR3	Added CSM enable bit <3>.
17772376 to 17772360	Kernel Data PAR	No difference.
17772356 to 17772340	Kernel Instruction PAR	No difference.
17772336 to 17772320	Kernel Data PDR	Added bypass cache, eliminated access flag and access modes other than 0, 2, and 6.
17772316 to 17772300	Kernel Instruction PDR	Added bypass cache, eliminated access flag and access modes other than 0, 2, and 6.
17772276 to 17772260	Supervisor Data PAR	No difference.
17772256 to 17772240	Supervisor Instruction PAR	No difference.
17772236 to 17772220	Supervisor Data PDR	Added bypass cache, eliminated access flag and access modes other than 0, 2, and 6.
17772216 to 17772200	Supervisor Instruction PDR	Added bypass cache, eliminated access flag and access modes other than 0, 2, and 6.

C.3 SOFTWARE DIFFERENCES

Table C-1 summarizes the programming differences (at the assembly language level) between the DCJ11 and other processors in the PDP-11 family.

	PROCESSORS													
ITEM	23/24	44	04	34	LSI11	05/10	15/20	35/40	45	70	60	J-11	T-11	VAX
1. OPR %R, (R) +; OPR %R, – (R) using the same register as both source and destination: contents of R are incremented (decremented) by 2 before being used as the source operand.	X						X	X			X	X	X	
OPR %R, (R) +; OPR %R, – (R) using the same register as both register and destination: initial contents of R are used as the source operand.		X	X	X	X	X			X	X				X
2. OPR %R, @ (R) +; OPR %R, @ – (R) using the same register as both source and destination: contents of R are incremented (decremented) by 2 before being used as the source operand.	X						X	X			X	X	X	
OPR %R, @ (R) +; OPR %R, @ – (R) using the same register as both source and destination: initial contents of R are used as the source operand.		X	X	X	X	X			X	X				X
3. OPR PC, X (R); OPR PC, @ X (R); OPR PC, @ A; OPR PC, A: location A will contain the PC of OPR +4.	X						X	X			X	X	X	
OPR PC, X (R); OPR PC, @ X (R), OPR PC, A; OPR PC, @ A: location A will contain the PC of OPR +2.		X	X	X	X	X			X	X				X
4. JMP (R) + or JSR reg, (R) +: contents of R are incremented by 2, then used as the new PC address.						X	X							
JMP (R) + or JSR reg, (R) +: initial contents of R are used as the new PC.	X	X	X	X	X			X	X	X	X	X	X	X

Table C-1 DCJ11 Programming Diffenences

ITEM	23/24	44	04	34	LSI11	05/10	15/20	35/40	45	70	60	J-11	T-11	VAX
5. JMP %R or JSR reg, %R traps to 10 (illegal instruction).	X		X	X	X	X	X	X			X		X	NA
JMP %R or JSR reg, %R traps to 4 (illegal instruction).		X							X	X		X		NA
6. SWAB does *not* change V.							X							
SWAB clears V.	X	X	X	X	X	X		X	X	X	X	X	X	X
7. Register addresses (177700–177717) are valid program addresses when used by CPU.						X							—[1]	—[1]
Register addresses (177700–177717) time out when used as a program address by the CPU. Can be addressed under console operation.		X	X	X			X	X	X	X	X			NA
Register addresses (177700–177717) time out when used as an address by CPU or console.	X				X							X		
8. Basic instructions noted in PDP-11 processor handbook.	X	X	X	X	X	X	X	X	X	X	X	X	X	X
SOB, MARK, RTT, SXT instructions*	X	X		X	X			X	X	X	X	X	X	—[2]
ASH, ASHC, DIV, MUL, XOR	X	X		X	X			X	X	X	X	X		X
Floating Point instructions in base machine.											X	X		
MFPT Instruction.	X	X										X		
The external option KE11-A provides MUL, DIV, SHIFT operation in the same data format.						X	X							

* RTT instruction is available in 11/04 but is different than other implementations.

[1] Register addresses (177700–177717) are handled as regular memory addresses in the I/O page.
[2] All but MARK.

ITEM	23/24	44	04	34	LSI11	05/10	15/20	35/40	45	70	60	J-11	T-11	VAX
The KE11-E (Expansion Instruction Set) provides the instructions MUL, DIV, ASH, and ASHC. These new instructions are 11/45 compatible.								X						
The KE11-F (Floating Instruction Set) adds unique stack ordered oriented point instructions: FADD, FSUB, FMUL, FDIV.								X						
The KEV-11 adds EIS/FIS instructions					X									
MFP, MTP instructions	X	X		X				X		X	X	X		
SPL Instruction		X							X	X		X		
CSM Instruction		X										X		
9. Power fail during RESET instruction is not recognized until after the instruction is finished (70 milliseconds). RESET instruction consists of 70 millisecond pause with INIT occurring during first 20 milliseconds.							X	X			X			
Power fail immediately ends the RESET instruction and traps if an INIT is in progress. A minimum INIT of 1 microsecond occurs if instruction aborted. PDP11-04/34/44 are similar with no minimum INIT time.		X	X	X					X	X				
Power fail acts the same as 11/45 (22 milliseconds with about 300 nanoseconds minimum). Power fail during RESET fetch is fatal with no power down sequence.						X								

ITEM	23/24	44	04	34	LSI11	05/10	15/20	35/40	45	70	60	J-11	T-11	VAX
RESET instruction consists of 10 microseconds of INIT followed by a 90 microsecond pause. Reset instruction consists of a minimum 8.4 microseconds followed by a minimum 100 nanosecond pause. Power fail not recognized until the instruction completes.	X				X							X		
10. No RTT instruction						X	X							
If RTT sets the "T" bit, the "T" bit trap occurs after the instruction following RTT.	X	X	X	X	X			X	X	X	X	X	X	X
11. If RTI sets "T" bit, "T" bit trap is acknowledged after instruction following RTI.						X	X							X
If RTI sets "T" bit, "T" bit trap is acknowledged immediately following RTI.	X	X	X	X	X			X	X	X	X	X	X	
12. If an interrupt occurs during an instruction that has the "T" bit set, the "T" bit trap is acknowledged before the interrupt.	X	X	X	X	X	X	X	X			X	X	X	NA[1]
If an interrupt occurs during an instruction and the "T" bit is set, the interrupt is acknowledged before "T" bit trap.									X	X				NA
13. "T" bit trap will sequence out of WAIT instruction.	X	X	X	X		X	X	X			X		X	NA
"T" bit trap will not sequence out of WAIT instruction. Waits until an interrupt.					X				X	X				

[1] Interrupts not visible to VAX compatibility mode.

ITEM	23/24	44	04	34	LSI11	05/10	15/20	35/40	45	70	60	J-11	T-11	VAX
14. Explicit reference (direct access) to PS can load "T" bit. Console can also load "T" bit.			X			X	X							
Only implicit references (RTI, RTT, traps and interrupts) can load "T" bit. Console cannot load "T" bit.	X	X		X	X			X	X	X	X	X	X	X
15. Odd address/non-existent references using the SP cause a HALT. This is a case of double bus error with the second error occurring in the trap servicing the first error. Odd address trap not implemented in LSI-11, 11/23 or 11/24.		X	X	X	X	X	X							
Odd address/non-existent references using the stack pointer cause a fatal trap. On bus error in trap service, new stack created at 0/2.	X							X	X	X	X	X	—[1]	—[2]
16. The first instruction in an interrupt routine will not be executed if another interrupt occurs at a higher priority level than assumed by the first interrupt.	X	X	X	X	X	X		X	X	X	X	X	X	X
The first interrupt in an interrupt service is guaranteed to be executed.							X							
17. Single general purpose register set implemented.	X	X	X	X	X	X	X	X			X		X	X
Dual general purpose register set implemented.									X	X		X		

[1] Odd address/non-existent references using SP do not trap.
[2] Odd address aborts to native mode.

ITEM	23/24	44	04	34	LSI11	05/10	15/20	35/40	45	70	60	J-11	T-11	VAX
18. PSW address, 177776, not implemented; must use instructions MTPS (move to PS) and MFPS (move from PS).					X								X	—[3]
PSW address implemented, MTPS and MFPS not implemented.		X	X			X	X	X	X	X	X			
PSW address and MTPS and MFPS implemented.	X			X								X		
19. Only one interrupt level (BR4) exists.					X									
Four interrupt levels exist.	X	X	X	X		X	X	X	X	X	X	X	X	NA
20. Stack overflow not implemented.					X								X	X
Some sort of stack overflow implemented.	X	X	X	X		X	X	X	X	X	X	X		
21. Odd address trap not implemented.	X				X								X	
Odd address trap implemented.		X	X	X		X	X	X	X	X	X	X		X
22. FMUL and FDIV instructions implicity use R6 (one push and pop); hence R6 must be set up correctly.					X									
FMUL and FDIV instructions do not implicitly use R6.								X						NA
23. Due to their execution time, EIS instructions can abort because of a device interrupt.					X									X
EIS instructions do not abort because of a device interrupt.	X	X		X				X	X	X	X	X		NA
24. Due to their execution time, FIS instructions can abort because of a device interrupt.					X			X						NA

[3] Can reference PSW only from native mode.

ITEM	23/24	44	04	34	LSI11	05/10	15/20	35/40	45	70	60	J-11	T-11	VAX
25. Due to their execution time, FP11 instructions can abort because of a device interrupt.*	X													
FP11 instructions do not abort because of a device interrupt.		X		X					X	X	X	X		NA
26. EIS instructions do a DATIP and DATO bus sequence when fetching source operand.					X									
EIS instructions do a DATI bus sequence when fetching source operand.	X	X		X				X	X	X	X	X		NA
27. MOV instruction does just a DATO bus sequence for the last memory cycle.	X	X		X	X			X	X	X	X	X		—[1]
MOV instruction does a DATIP and DATO bus sequence for the last memory cycle.			X			X	X						—[2]	
28. If PC contains non-existent memory and a bus error occurs, PC will have been incremented.	X	X	X	X	X	X	X		X	X		X		
If PC contains non-existent memory address and a bus error occurs, PC will be unchanged.								X					—[3]	X
29. If register contains non-existent memory address in mode 2 and a bus error occurs, register will be incremented.	X				X	X	X	X	X	X		X	—[3]	
Same as above but register is unchanged.		X	X	X										

* Integral floating point assumed on 11/23 and 11/24; FP11E assumed for 11/60.
[1] Implementation dependent.
[2] MOV instruction does a DATI and a DATO bus sequence for last memory cycle.
[3] Does not support bus errors.

ITEM	23/24	44	04	34	LSI11	05/10	15/20	35/40	45	70	60	J-11	T-11	VAX
30. If register contains an odd value in mode 2 and a bus error occurs, register will be incremented.	X				X			X	X	X		X	—[3]	—[4]
If register contains an odd value in mode 2 and a bus error occurs, register will be unchanged.		X	X	X		X	X							
31. Condition codes restored to original values after FIS interrupt abort (EIS doesn't abort on 35/40).								X						
Condition codes that are restored after EIS/FIS interrupt abort are indeterminate.					X									NA
32. Opcodes 075040 through 075377 unconditionally trap to 10 as reserved opcodes.	X	X	X	X		X	X	X	X	X	X	X	X	—[1]
If KEV-11 option is present, opcodes 75040 through 07533 perform a memory read using the register specified by the low order 3 bits as a pointer. If the register contents are a non-existent address, a trap to 4 occurs. If the register contents are an existent address, a trap to 10 occurs.					X									
33. Opcodes 210 thru 217 trap to 10 as reserved instructions.	X	X	X	X		X	X	X	X	X	X	X	X	—[1]
Opcodes 210 thru 217 are used as a maintenance instruction.					X									

[3] Does not support bus errors.
[4] Unpredictable.
[1] Traps to native mode.

ITEM	23/24	44	04	34	LSI11	05/10	15/20	35/40	45	70	60	J-11	T-11	VAX
34. Opcodes 75040 thru 75777 trap to 10 as reserved instructions.	X	X	X	X		X	X	X	X	X	X	X	X	—[1]
If KEV-11 options is present, opcodes 75040 thru 75577 can be used as escapes to user microcode. If no user microcode exists, a trap to 10 occurs.					X									
35. Opcodes 170000 thru 177777 trap to 10 as reserved instructions.			X			X	X	X					X	—[1]
Opcodes 170000 thru 177777 are implemented as floating point instructions.	X	X		X					X	X	X	X		
Opcodes 170000 thru 177777 can be used as escapes to user microcode. If no user microcode exists, a trap to 10 occurs.					X									
Opcode 076600 used for maintenance.											X			
36. CLR and SXT do just a DATO sequence for the last bus cycle.	X											X		—[1]
CLR and SXT do DATIP-DATO sequence for the last bus cycle.		X	X	X	X	X	X	X	X	X	X		—[2]	
37. MEM MGT maintenance mode MMR0 bit 8 is implemented.		X		X				X	X	X	X			
MEM MGT maintenance mode MMR0 bit 8 is not implemented.	X											X		NA
38. PS<15:12>, non-kernel mode, non-kernel stack pointer and MTPx and MFPx instructions exist even when MEM MGT is not configured.	X	X							X	X	X	X		

[1] Traps to native mode.

[1] Unpredictable.
[2] CLR and SXT do DATI-DATO.

ITEM	23/24	44	04	34	LSI11	05/10	15/20	35/40	45	70	60	J-11	T-11	VAX
PS<15:12>, non-kernel mode, non-kernel stack pointer, and MTPx and MFPx instructions exist only when MEM MGT is configured.								X						NA
39. Current mode PS bits <15:14> set to 01 or 10 will cause a MEM MGT trap upon any memory reference.	X			X				X						
Current mode PS bits <15:14> set to 10 will be treated as kernel mode (00) and not cause a MEM MGT trap.	X													NA
Current mode PS bits <15:14> set to 10 will cause a MEM MGT trap upon any memory reference.		X							X	X		X		
40. MTPS in user mode will cause MEM MGT trap if PS address 177776 not mapped. If mapped, PS <7:5> and <3:0> affected.				X										
MTPS in non-user mode will not cause MEM MGT trap and will only affect PS <3:0> regardless of whether PS address 177776 is mapped.	X											X		NA
41. MFPS in user mode will cause MEM MGT if PS address 177776 not mapped. If mapped, PS <7:0> are accessed.				X										
MTPS in user mode will not trap regardless of whether PS address 177776 is mapped.	X											X		NA

[1] Unpredictable.
[2] CLR and SXT do DATI-DATO.

ITEM	23/24	44	04	34	LSI11	05/10	15/20	35/40	45	70	60	J-11	T-11	VAX
42. Programs cannot execute out of internal processor registers.												X		
Programs can execute out of internal processor registers.	X	X		X				X	X	X	X			
43. A HALT instruction in user or supervisor mode will trap thru location 4.		X							X	X		X		
A HALT instruction in user or supervisor mode will trap thru location 10.	X			X				X			X		—[1]	—[2]
44. PDR bit <0> implemented.									X	X				
PDR bit <0> not implemented.	X	X		X				X			X	X	X	X
45. PDR bit <7> (any access) implemented.									X	X				
PDR bit <7> (any access) not implemented.	X	X		X				X			X	X	X	X
46. Full PAR <15:0> implemented.	X	X								X		X		
Only PAR <11:0> implemented.				X				X	X		X		X	X
47. MMR0<12>—trap-memory management—implemented.									X	X				
MMR0<12> not implemented.	X	X		X				X			X	X	X	X
48. MMR3<2:0>—D space enable—implemented.		X							X	X		X		
MMR3<2:0> not implemented.	X			X				X			X		X	X
49. MMR3<5:4>—IOMAP, 22-bit mapping enabled—implemented.	X	X								X		X		
MMR3<5:4> not implemented.				X				X	X		X		X	X

[1] HALT pushes PC & PSW to stack, loads PS with 340 and PC with <powerup address> + 40.
[2] Traps to native mode.

ITEM	23/24	44	04	34	LSI11	05/10	15/20	35/40	45	70	60	J-11	T-11	VAX
50. MMR3<3>–CSM enable–implemented.		X										X		
MMR3<3> not implemented.	X			X				X	X	X	X		X	X
51. MMR2 tracks instruction fetches and interrupt vectors.									X	X				
MMR2 tracks only instruction fetches.	X	X		X				X			X	X	NA	NA
52. MFPx %6, MTPx when PS<13:12>= 10 gives unpredictable results.	X	X		X				X	X	X	X			
MTPx %6, MTPx %6 when PS<13:12>= 10 uses user stack pointer.												X	NA	NA

[1] HALT pushes PC & PSW to stack, loads PS with 340 and PC with <powerup address> + 40.
[2] Traps to native mode.

APPENDIX D
INSTRUCTION TIMING

The execution time for an instruction depends on: (1) the type of instruction executed, (2) the the mode of addressing used, and (3) the type of memory being referenced. In general, the total execution time is the sum of the base instruction fetch/execute time plus the operand(s) address calculation/fetch time.

The tables in this appendix can be used to calculate the length of an instruction in terms of microcycles (MC). In the first group of tables, the first column specifies the number of microcycles required to fetch/execute the base instruction. The R/W column specifies how many of these microcycles are read microcycles and how many are write microcycles (any remaining microcycles are NIO). If the instruction involves the calculation/fetch of one or more operands, a reference to a separate table (a source or destination table) is made in the last column(s). The source/destination tables reveal how many microcycles the source/destination calculation/fetch takes and also specifies how many of these are read or write microcycles (again, any remaining microcycles are NIO).

The numbers in the tables are based on the assumption that a memory read must last a minimum of four CLK periods, a memory write must last a minimum of eight CLK periods, and an NIO lasts four CLK periods (no DMA). Any wait states caused by slower memory or a DMA transfer must be added to the total instruction time. If wait states are required, the first wait state of a non-stretched read or NIO cycle will last four clock periods, and can continue in increments of two clock periods. Further wait states for stretched cycles occur in increments of two clock periods.

Floating-point instruction execution times are given as a range. The actual execution time will vary depending on the type of data being operated on.

Here are two examples of how to use the tables:

Example 1:

How long does a MOV R0,@#2044 instruction last?

Step 1: From the tables, the execution time for the MOV base instruction is found to be 1 microcycle (MC), or four CLK periods. This consists of one read and no write microcycles. Depending upon the type of memory in the system, the microcycle may be stretched. If so, the microcycle lasts at least eight CLK periods and may be stretched thereafter in increments of two CLK periods.

Step 2: To find the operand calculation/fetch time for the source operand (R0), refer to Table S1. From Table S1, it is seen that a mode 0 register 0 calculate/fetch takes 0 microcycles. Note that the operand is already available to the DCJ11 (in the register file).

Step 3: To find the operand calculation/fetch time for the destination operand (the contents of memory location 2044), refer to Table D3. From Table D3, it is seen that a mode 3 register 7 calculate/fetch takes 3 microcycles, one of which is a read microcycle and one of which is a write microcycle. Note that the remaining microcycle is an NIO microcycle. Once again, the type of memory in the system must be taken into account. If the read cycle is stretched, the stretched cycle lasts at least eight CLK periods and may be stretched thereafter in increments of two CLK periods. The write microcycle lasts at least eight CLK periods and may be stretched in increments of two CLK periods.

Step 4: For a determination of the minimum time required, total up the microcycles. In this example, It is 1 + 0 + 3, or 4 microcycles (which is 16 CLK periods if no microcycle stretching occurs).

Example 2:

The source and destination tables for floating point instructions show a negative number in the MC column for certain mode 2 register 7 operations. This example shows a timing calculation for one of these.

How long does an CLRD #2000 instruction last?

Step 1: The base instruction time for the CLRD instruction is 14 microcycles.

Step 2: From Table F2, the calculation/fetch time for the operand (a mode 2 register 7 reference) is shown as (-1). This means that one microcycle should be subtracted from the base instruction time. However, add one microcycle for the memory write operation. There are no memory read cycles to account for.

Step 3: Total up the microcycles: 14 - 1 + 1 = 14 microcycles minimum (assumes no cycle stretching).

SINGLE OPERAND

Mnemonic	Instruction	TIMING Execution MC	R/W	Source Table	Dest Table
General					
CLR(B)	Clear	1	1/0	—	D3
COM(B)	Complement (1's)	1	1/0	—	D4
INC(B)	Increment	1	1/0	—	D4
DEC(B)	Decrement	1	1/0	—	D4
NEG(B)	Negate (2's complement)	1	1/0	—	D4
TST(B)	Test	1	1/0	—	D4
Rotate and Shift					
ROR(B)	Rotate right	1	1/0	—	D4
ROL(B)	Rotate left	1	1/0	—	D4
ASR(B)	Arithmetic shift right	1	1/0	—	D4
SWAB	Swap bytes	1	1/0	—	D4
Multiple-Precision					
ADC(B)	Add carry	1	1/0	—	D4
SBC(B)	Subtract carry	1	1/0	—	D4
SXT	Sign extend	1	1/0	—	D3
Multiprocessing					
TSTSET	Test and set (low bit interlocked)	5	1/1	—	D4
WRTLCK	Write interlocked	4	1/1	—	D4

DOUBLE OPERAND

Mnemonic	Instruction	TIMING Execution MC	R/W	Source Table	Dest Table
General					
MOV(B)	Move	1	1/0	S1	D3
CMP(B)	Compare	1	1/0	S1	D2
ADD	Add	1	1/0	S1	D4
SUB	Subtract	1	1/0	S1	D4
Logical					
BIT(B)	Bit test (AND)	1	1/0	S1	D2
BIC(B)	Bit clear	1	1/0	S1	D4
BIS(B)	Bit set (OR)	1	1/0	S1	D4

Register

Mnemonic	Instruction					
MUL	Multiply	22	1/0	—	D1	(Notes 5,11)
DIV	Divide	34	1/0	—	D1	(Notes 6,7,12)
ASH	Shift automatically	4	1/0	—	D1	
ASHC	Arith shift combined	5	1/0	—	D1	(Note 13)
XOR	Exclusive (OR)	1	1/0	—	D4	

BRANCH

Mnemonic	Instruction	TIMING: Branch Not Taken MC	R/W	Branch Taken MC	R/W
Branches					
BR	Branch (unconditional)	2	1/0	4	2/0
BNE	Br if not equal (to 0)	2	1/0	4	2/0
BEQ	Br if equal (to 0)	2	1/0	4	2/0
BPL	Br if plus	2	1/0	4	2/0
BMI	Br if minus	2	1/0	4	2/0
BVC	Br if overflow is clear	2	1/0	4	2/0
BVS	Br if overflow is set	2	1/0	4	2/0
BCC	Br if carry is clear	2	1/0	4	2/0
BCS	Br if carry is set	2	1/0	4	2/0
Signed Conditional Branches					
BGE	Br if greater or equal (to 0)	2	1/0	4	2/0
BLT	Br if less than (0)	2	1/0	4	2/0
BGT	Br if greater than (0)	2	1/0	4	2/0
BLE	Br if less or equal (to 0)	2	1/0	4	2/0

Mnemonic	Instruction	Branch Not Taken MC	R/W	Branch Taken MC	R/W
Unsigned Conditional Branches					
BHI	Branch if higher	2	1/0	4	2/0
BLOS	Branch if lower or same	2	1/0	4	2/0
BHIS	Branch if higher or same	2	1/0	4	2/0
BLO	Branch if lower	2	1/0	4	2/0
SOB	Subtract 1 and branch (if ≠ 0)	3	1/0	5	2/0

JUMP and SUBROUTINE

Mnemonic	Instruction	TIMING: Execution MC	R/W	DST Table	
JMP	Jump	—	—	D5	
JSR	Jump to subroutine	—	—	D6	(Note 4)
RTS	Return from subroutine	5	3/0	—	(Note 14)
MARK	Stack cleanup	10	3/0		

TRAP and INTERRUPT		TIMING	
		Execution	
Mnemonic	Instruction	MC	R/W
EMT	Emulator trap	20	4/2
TRAP	Trap	20	4/2
BPT	Breakpoint trap	20	4/2
IOT	Input/output trap	20	4/2
RTI	Return from interrupt	9	4/0
RTT	Return from interrupt	9	4/0

CONDITION CODE OPERATORS		TIMING	
		Execution	
Mnemonic	Instruction	MC	R/W
CLC	Clear C	3	1/0
CLV	Clear V	3	1/0
CLZ	Clear Z	3	1/0
CLN	Clear N	3	1/0
CCC	Clear all CC bits	3	1/0
SEC	Set C	3	1/0
SEV	Set V	3	1/0
SEZ	Set Z	3	1/0
SEN	Set N	3	1/0
SCC	Set all CC bits	3	1/0

MISCELLANEOUS		TIMING		
		Execution		Dest
Mnemonic	Instruction	MC	R/W	Table
HALT	Halt	-		—
WAIT	Wait for interrupt	-		—
RESET	Reset external bus	-		—
NOP	(No operation)	3	1/0	—
SPL	Set priority level to N	7	1/0	—
MFPI	Move from previous instr space	5	1/1	D1
MTPI	Move to previous instr space	3	2/0	D3
MFPD	Move from previous data space	5	1/1	D1
MTPD	Move to previous data space	3	2/0	D3
MTPS	Move byte to PSW PS ← (svc)	8	1/0	D1
MFPS	Move byte from PSW (dst) ← PS <7:0>	1	1/0	D3
MFPT	Move from processor (R0<7:0><proc code	2	1/0	—
CSM	Call to supervisor mode	28	3/3	D1

FLOATING POINT

Mnemonic	Instruction	TIMING Execution (MC) Min	Typ	Max	Non Mode 0 Table
ABSD	Make Absolute	23		24	F3
ABSF	Make Absolute	19		20	F3
ADDD	Add	41	48	119	F1
ADDF	Add	31	35	102	F1
CFCC	Copy Floating Condition Codes	5		5	—
CLRD	Clear	14		14	F2
CLRF	Clear	12		12	F2
CMPD	Compare	24		25	F1
CMPF	Compare	18		19	F1
DIVD	Divide	160		167	F1
DIVF	Divide	59		63	F1
LDCDF	Ld & C from D to F	24		26	F1
LDCFD	Ld & C from F to D	20		21	F1
LDCID	Ld & C Integer to D	31		42	F4
LDCIF	Ld & C Integer to F	26		36	F4
LDCLD	Ld & C Long Integer to D	31		52	F4
LDCLF	Ld & C Long Integer to F	26		44	F4
LDD	Load	16		17	F1
LDEXP	Load Exponent	17		18	F4
LDF	Load	12		13	F1
LDFPS	Load FPP Program Status	6		6	F4
MODD	Multiply and Separate	202	217	268	F1
MODF	Integer and Fraction	82	94	115	F1
MULD	Multiply	165		173	F1
MULF	Multiply	56		61	F1
NEGD	Negate	22		23	F3
NEGF	Negate	18		19	F3
SETD	Set Floating Double Mode	6		6	—
SETF	Set Floating Mode	6		6	—
SETI	Set Integer Mode	6		6	—
SETL	Set Long Integer Mode	6		6	—
STCDF	St & C from D to F	17		20	F2
STCDI	St & C from D to Integer	26		38	F5
STCDL	St & C from D to Long Integer	26		54	F5
STCFD	St & C from F to D	19		20	F2
STCFI	St & C from F to Integer	23		35	F5
STCFL	St & C from F to Long Integer	23		51	F5
STD	Store	12		12	F2
STECP	Store Exponent	16		16	F5
STF	Store	8		8	F2
STFPD	Store FPP Program Status	9		9	F5
STST	Store FPP Status	7		7	F5
SUBD	Subtract	47	55	122	F1
SUBF	Subtract	37	41	104	F1
TSTD	Test	11		12	F1
TSTF	Test	9		10	F1

SOURCE AND DESTINATION TABLES:

Table S1 Source Address Time: All Double Operand

Source Mode	Source Register	Microcode Cycles	Read Memory Cycles	
0	0-7	0	0	
1	0-7	2	1	
2	0-6	2	1	
2	7	1	1	
3	0-6	4	2	
3	7	3	2	
4	0-6	3	1	
4	7	6	2	(Note 1)
5	0-6	5	2	
5	7	8	3	(Note 1)
6	0-7	4	2	
7	0-7	6	3	

Table D1 Destination Address Time: Read Only Single Operand

Destination Mode	Destination Register	Microcode Cycles	Read Memory Cycles	
0	0-7	0	0	
1	0-7	2	1	
2	0-6	2	1	
2	7	1	1	
3	0-6	4	2	
3	7	3	2	
4	0-6	3	1	
4	7	7	2	(Note 2)
5	0-6	5	2	
5	7	9	3	(Note 3)
6	0-7	4	2	
7	0-7	6	3	

Table D2 Destination Address Time: Read Only Double Operand

Destination Mode	Destination Register	Microcode Cycles	Read Memory Cycles	
0	0-7	0	0	
1	0-7	3	1	
2	0-6	3	1	
2	7	2	1	
3	0-6	5	2	
3	7	4	2	
4	0-6	4	1	
4	7	8	2	(Note 2)
5	0-6	6	2	
5	7	10	3	(Note 3)
6	0-7	5	2	
7	0-7	7	3	

Table D3 Destination Address Time: Write Only

Destination Mode	Destination Register	Microcode Cycles	Memory Cycles Read	Write
0	0-6	0	0	0
0	7	5	1	0
1	0-6	2	0	1
1	7	6	1	1
2	0-6	2	0	1
2	7	6	1	1
3	0-6	4	1	1
3	7	3	1	1
4	0-6	3	0	1
4	7	7	1	1
5	0-6	5	1	1
5	7	9	2	1
6	0-7	4	1	1
7	0-7	6	2	1

Table D4 Destination Address Time: Read Modify Write

Destination Mode	Destination Register	Microcode Cycles	Memory Cycles Read	Write	
0	0-6	0	0	0	
0	7	5	1	0	
1	0-6	3	1	1	
1	7	7	2	1	
2	0-6	3	1	1	
2	7	7	2	1	
3	0-6	5	2	1	
3	7	4	2	1	
4	0-6	4	1	1	
4	7	8	2	1	(Note 2)
5	0-6	6	2	1	
5	7	10	3	1	(Note 3)
6	0-7	5	2	1	
7	0-7	7	3	1	

Table D5 Destination Address Time: JMP

Destination Mode	Destination Register	Microcode Cycles	Memory Cycles Read	Write
1	0-7	4	2	0
2	0-7	6	2	0
3	0-7	5	3	0
4	0-7	5	2	0
5	0-7	6	3	0
6	0-6	6	3	0
6	7	5	3	0
7	0-7	7	4	0

Table D6 Destination Address Time: JSR

Destination Mode	Destination Register	Microcode Cycles	Memory Cycles Read	Write
1	0-7	9	2	1
2	0-7	10	2	1
3	0-6	10	3	1
3	7	9	3	1
4	0-7	10	2	1
5	0-7	11	3	1
6	0-6	10	3	1
6	7	9	3	1
7	0-7	12	4	1

Table F1 Floating Source Modes 1-7

Single Precision

Mode	Register	Microcode Cycles	Memory Read	Memory Write
1	0-7	3	2	0
2	0-6	3	2	0
2	7	1	1	0
3	0-6	4	3	0
3	7	3	3	0
4	0-7	4	2	0
5	0-7	5	3	0
6	0-7	4	3	0
7	0-7	6	4	0

Double Precision

Mode	Register	Microcode Cycles	Memory Read	Memory Write
1	0-7	5	4	0
2	0-6	5	4	0
2	7	0 (Note 15)	1	0
3	0-6	6	5	0
3	7	5	5	0
4	0-7	6	4	0
5	0-7	7	5	0
6	0-7	6	5	0
7	0-7	8	6	0

Table F2 Floating Destination Modes 1-7

Single Precision

Mode	Register	Microcode Cycles	Memory Read	Memory Write
1	0-7	3	0	2
2	0-6	3	0	2
2	7	1	0	1
3	0-6	4	1	2
3	7	3	1	2
4	0-7	4	0	2
5	0-7	5	1	2
6	0-7	4	1	2
7	0-7	6	2	2

Double Precision

Mode	Register	Microcode Cycles	Memory Read	Memory Write
1	0-7	5	0	4
2	0-6	5	0	4
2	7	(-1) (Note 15)	0	1
3	0-6	6	1	4
3	7	5	1	4
4	0-7	6	0	4
5	0-7	7	1	4
6	0-7	6	1	4
7	0-7	8	2	4

Table F3 Floating Read Modify Write Modes 1-7

Single Precision

Mode	Register	Microcode Cycles	Memory Read	Memory Write
1	0-7	5	2	2
2	0-6	5	2	2
2	7	1 (Note 15)	1	1
3	0-6	6	3	2
3	7	5	3	2
4	0-7	6	2	2
5	0-7	7	3	2
6	0-7	6	3	2
7	0-7	8	4	2

Table F3 Floating Read Modify Write Modes 1-7

Double Precision

Mode	Register	Microcode Cycles	Memory Read	Memory Write
1	0-7	9	4	4
2	0-6	9	4	4
2	7	(-2) (Note 15)	1	1
3	0-6	10	5	4
3	7	9	5	4
4	0-7	10	4	4
5	0-7	11	5	4
6	0-7	10	5	4
7	0-7	12	6	4

Table F4 Integer Source Modes 1-7

Integer

Mode	Register	Microcode Cycles	Memory Read	Memory Write
1	0-7	2	1	0
2	0-6	2	1	0
2	7	0 (Note 15)	1	0
3	0-6	3	2	0
3	7	2	2	0
4	0-7	3	1	0
5	0-7	4	2	0
6	0-7	3	2	0
7	0-7	5	3	0

Long Integer

Mode	Register	Microcode Cycles	Memory Read	Memory Write
1	0-7	4	2	0
2	0-6	4	2	0
2	7	0 (Note 15)	1	0
3	0-6	5	3	0
3	7	4	3	0
4	0-7	5	2	0
5	0-7	6	3	0
6	0-7	5	3	0
7	0-7	7	4	0

Table F5 Integer Destination Modes 1-7

Integer

Mode	Register	Microcode Cycles	Memory Read	Memory Write
1	0-7	2	0	1
2	0-6	2	0	1
2	7	2	0	1
3	0-6	3	1	1
3	7	2	1	1
4	0-7	3	0	1
5	0-7	4	1	1
6	0-7	3	1	1
7	0-7	5	2	1

Long Integer

Mode	Register	Microcode Cycles	Memory Read	Memory Write
1	0-7	4	0	2
2	0-6	4	0	2
2	7	2	0	1
3	0-6	5	1	2
3	7	4	1	2
4	0-7	5	0	2
5	0-7	6	1	2
6	0-7	5	1	2
7	0-7	7	2	2

NOTES

1. Subtract 2 microcycles (MC) and one read if both source and destination modes autodecrement PC, or if WRITE-ONLY or READ-MODIFY-WRITE mode 07 or 17 is used.

2. READ-ONLY and READ-MODIFY-WRITE destination mode 47 references actually perform 3 READ operations. For bookkeeping purposes, one of the READs is accounted for in the EXECUTE, FETCH TIMING.

3. READ-ONLY and READ-MODIFY-WRITE destination mode 57 references actually perform 4 READ operations. For bookkeeping purposes, one of the READs is accounted for in the EXECUTE, FETCHING TIMING.

4. Subtract 1 MC if the link register is PC.

5. Add 1 MC if the source operand is negative.

6. Subtract 1 MC if the source mode is not zero.

7. Add 1 MC if the quotient is even.
   Add 2 MC if overflow occurs.
   Add 5 MC and 1 read if the PC is used as a destination register, but only if source mode 47 or 57 is not used.

8. Add 1 MC per shift.

9. Add 1 MC if source operand <15:6> is not zero.

10. Subtract 1 MC if one shift only.

11. Add 4 MC and 1 read if the PC is used as a destination register, but only if source mode 47 or 57 is not used.

12. Divide by zero executes in 5 MC (see note 6).

13. Timing for no shift. Add 1 MC if a left shift. (Notes 8, 9, 11 apply.) Add 2 MC for a right shift. (Notes 8, 10, 11 apply.)

14. Add one MC if a register other than R7 is used.

15. Mode 27 references only access single word operands. The execution time listed has been compensated in order to accurately compute the total execution time.

Bus lock -
An indication to memory to prevent or "lock" out other accesses to that location until it is unlocked. This occurs during an RMW read bus microcycle with the bus lock control bit asserted. Memory is automatically unlocked by the following Bus Write microcycle by that processor.

Cache bypass -
Unconditionally bypass cache and access main memory directly. If the cache entry is valid, typically invalidate it.

Cache force miss -
Unconditionally bypass cache and access main memory directly. If the cache entry is valid, typically do not invalidate it but ignore it.

Data stream bus cycle -
Any microcycle which is a Read, Read/Modify/Write or Write microcycle.

Demand abort -
An abort during a demand bus microcycle.

Instruction stream bus cycle -
Any microcycle which is a prefetch microcycle.

Internal registers -
These explicitly addressable registers are the PS, PIRQ, MMR0, MMR1, MMR2, MMR3, Hit/Miss, CPU Error, PARs, and PDRs.

Predecode -
An indication to decode the next PDP-11 instruction. This occurs during a microcycle in which the DCJ11 asserts PRDC and decodes the prefetch buffer contents as the next PDP11 instruction.

Read/Modify/Write (RMW) operation -
Two consecutive microcycles in which the first is a Bus Read microcycle and the second is a Bus Write microcycle. Both microcycles access the same location.

Request abort -
An abort during a request bus microcycle. If it is a memory management or address abort, it will not stretch the microcycle.

INDEX

Digital Equipment Corporation • Bedford, MA 01730